

Eclipse Plug-Ins

Third Edition

Eclipse插件开发

(原书第3版)

(美) Eric Clayberg 著
Dan Rubel

陈沛 等译



机械工业出版社
China Machine Press

Eclipse插件开发 (原书第3版)

创建商业质量的插件意味着要比集成至Eclipse的最小要求做得更多、更远。它意味着需要处理商业版本“得体与优雅”所要求的所有细节。本书包含了插件开发的全过程,包括达到最高质量所必需的所有额外步骤。

本书是全球畅销书,而作者完全重写了第3版以体现Eclipse 3.4各项强大的新功能。业界领先的Eclipse专家Eric Clayberg和Dan Rubel展示了插件开发所有细节和实践经验,并对开发者很有可能遇到的问题给出有针对性的、经过实践证明的解决方案。

本书已经完全更新了所有代码示例、相关API列表、图标和屏幕截图以体现Eclipse 3.4 API和最新的Java语法。此外,Clayberg和Rubel已经完全更新了广大受欢迎的Favorite用例,重新编写了大部分内容和代码。作者慎重地展开对已有Eclipse功能(比如视图和编辑器)的附加新功能的讨论,并详细解释了新的功能(如命令、GEF和PDE构建)。

本书主要内容

- 完整包含了Eclipse的新内容。
- 阐述了强大的Eclipse命令框架。该命令框架取代较老的Eclipse操作框架。
- 给出关于通过视图和编辑器使用命令的相关讨论。
- 介绍了Mylyn。它是一个降低信息过载和减轻多任务复杂度的专注于任务的接口。
- 包含关于图形编辑框架(Graphical Editing Framework, GEF)的全新的一章。GEF可以用于创建动态的、交互的图形用户界面元素。
- 带领你浏览PDE构建过程的每一个步骤。
- 展示如何使用p2创建更新站点。使用p2代替较老的更新管理器。

本书适用于对扩展Eclipse平台、Rational软件开发平台或支持Eclipse插件的其他平台感兴趣的有经验的开发者。

作者简介

Eric Clayberg 是Instantiations公司产品开发部高级副总裁。Eric是一位具有丰富经验的软件技术专家、产品开发人员、企业家和具有超过17年商业软件开发经验的项目经理。他拥有麻省理工学院的理学学士学位、哈佛大学的MBA学位。他是两家成功的软件公司ObjectShare和Instantiations的共同创始人之一。



Dan Rubel Instantiations公司的首席技术执行官。他是一名成功的商人,也是面向对象技术方面的设计与应用专家。他具有十五年以上的商业软件开发经验。他拥有Bucknell的理学学士学位。他是Instantiations公司的创始人之一。



Instantiations是IBM高级商业伙伴,为Eclipse和IBM的VisualAge、WebSphere和Rational产品系列开发了許多商业附加软件。



客服热线: (010) 88378991, 88361066
购书热线: (010) 68326294, 88379649, 68995259
投稿热线: (010) 88379604
读者信箱: hzsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

www.pearsonhighered.com

上架指导: 计算机 程序设计

ISBN 978-7-111-30336-7



9 787111 303367

定价: 85.00元

开发人员专业技术丛书

Eclipse Plug-Ins

Third Edition

Eclipse插件开发

(原书第3版)

(美) Eric Clayberg 著
Dan Rubel

陈沛 等译

机械工业出版社
China Machine Press

本书是一本由两位长期从事Java商业软件开发的技术专家编写的关于开发Eclipse商业插件的指南。本书主要介绍了开发Eclipse商业插件的完整过程,并从标准窗口小部件工具集、命令与操作、视图、透视图、实现帮助、国际化等方面对创建Eclipse商业插件进行了详细描述。本书既包含了开发Eclipse插件的基础理论,也涵盖了大量关于Eclipse插件开发的细节。

本书不仅适用于Eclipse插件开发的初学者,对于Eclipse商业软件开发人员也有很高的参考价值。

Simplified Chinese edition copyright © 2010 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Eclipse Plug-Ins, 3E* (ISBN 978-0-321-55346-1) by Eric Clayberg, Dan Rubel, Copyright © 2009.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有Pearson Education(培生教育出版集团)激光防伪标签,无标签者不得销售。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2009-6761

图书在版编目(CIP)数据

Eclipse插件开发(原书第3版)/(美)克莱伯格(Clayberg, E.), (美)鲁贝(Rubel, D.)著;陈沛等译. —北京:机械工业出版社, 2010.5

(开发人员专业技术丛书)

书名原文: *Eclipse Plug-Ins, Third Edition*

ISBN 978-7-111-30336-7

I. E… II. ①克… ②鲁… ③陈… III. 软件工程—程序设计 IV. TP311.56

中国版本图书馆CIP数据核字(2010)第063064号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:李东震

北京瑞德印刷有限公司印刷

2010年5月第1版第1次印刷

186mm × 240mm · 36印张

标准书号: ISBN 978-7-111-30336-7

定价: 85.00元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88379649; 68995259

投稿热线: (010) 88379604

读者信箱: hzjsj@hzbook.com



读者赞誉

“我经常被问到，‘有哪些关于Eclipse最好的书？’在我每一次的回答中，第一本都是本书。在我看来，对于全世界的软件开发人员来说，它是Eclipse书籍中最清晰、最切合主题的。其他Eclipse书籍往往关注于Eclipse的内部基础结构或重复Eclipse文档，而该书像激光一样地专注主题，并专注于当你想要创建软件产品所需要考虑的东西。”

——Bjorn Freeman-Benson
Eclipse基金会开源组织负责人

“这本大部头的书试图成为开发Eclipse插件的最佳实践指南。在我看来，在这一方面，它做得很成功。在你想要分发你所开发的插件之前，请看看这本书。”

——Ernest Friedman-Hill
JavaRanch.com主管

“如果你想找一本Eclipse插件开发相关书籍作为指南，那就是本书了。虽然还有其他关于Eclipse的书籍，但很少有像本书这么深入的。”

——Simon Archer

“本书对我们团队的每一名成员都是非常宝贵的培训辅导资料。事实上，如果不用本书作为基础，对我们团队进行培训是不可能完成的。现在它成为我们所有开发人员的必读书籍，并且帮助我们准时地、不超出预算地交付了一个全新的、非常复杂的产品。感谢本书在阐述开发Eclipse插件过程中所做的伟大工作。”

——Bruce Gruenbaum

“本书无疑是所有我的书籍中最有用的书之一。如果你准备开始开发Eclipse插件，本书是必备书籍之一。它将节省你大量的时间和精力。在本书中，你将会发现很多有用的建议，尤其是那些有益于提升专业技巧和提高插件的完整性的。本书特色鲜明、结构良好、考虑周到、写作清晰，并且不包含任何多余内容。用于描述不同组件和清单的章节关系的图表是很好的，为理解各部分如何协同工作提供了帮助。本书还包含了操作、视图和编辑器等相关内容。我相信每个读者都能从作者的经验中受益。当然我就是受益者之一。”

——Tony Saveski

“这本有重要影响的书的作者拥有数十年的经验。这些经验是有关于目前已知的最具生产效率和最健壮的软件工程的。他们的经验现在已经成功地应用于使用Eclipse进行更有效的Java开发。这是一本每一个专业软件工程开发人员都应该拥有的书。”

——Ed Klimas

“仅仅是想让你们知道这是一本优秀的书。同时也感谢作者为创作这样一本易懂而又专业的书所做的努力。”

——Brooke Hedrick

“为Eclipse开发良好插件的关键是懂得在何处，以及怎样扩展该IDE。这就是本书所要告诉你的。它是专业插件开发者的必备，尤其是那些开发商业应用的人。我不能没有它。”

——Brian Wilkerson



译者序

Eclipse是一个开源的可扩展开发平台。它是一个用于构建、部署和管理软件生命周期的运行时的平台和应用程序框架。现今，有越来越多的软件开发人员选择使用Eclipse，不仅仅是因为它的开源特性，更重要的在于它的可扩展性。

我们可以通过向Eclipse添加插件从而扩展它的功能。现在，有大量的插件几乎可以让Eclipse完成你所能想到的任何软件开发任务。因此，为Eclipse开发良好的插件就显得十分重要。

本书的两位作者有十几年的商业软件开发经验，对于如何开发结构良好、质量可靠的Eclipse插件有着十分丰富的实践经验。因此，通过阅读本书，读者可以获取关于开发Eclipse商业级质量插件的、经过实践检验的相关知识。

本书首先对Eclipse结构进行了介绍，然后通过创建并不断完善一个名为收藏夹的示例插件，阐述了开发Eclipse插件的全方位的内容。本书的结尾是关于Eclipse插件开发的一些高级话题，并列举了一些在Eclipse社区有良好口碑的商业插件。

本书英文第3版是基于Eclipse 3.4的，而翻译本书时Eclipse的最新版本为Eclipse 3.5 Galileo。因此，在使用Eclipse 3.5开发或运行本书中的示例时，可能会存在一些区别。

参加本书翻译的人员有：陈沛、袁芳、伍汉和、韩璐、强永刚、许飞、李婧、刘依承、贺娟娟、陈服兵、郭爱华。

最后，虽然译者尽了最大努力想要完成好本书的翻译，试图做到准确、符合中文语言习惯，但由于译者水平有限，译文肯定存在一些不足之处。如果您发现了不足之处并能通过邮件向译者指出，译者将不胜感激！邮箱地址为andyflying2007@gmail.com。

陈 沛

2010年3月



序 言

对于全世界数百万的开发人员、工程师和用户来说，Eclipse是一个用于工具集成的可扩展平台。对于全世界数十万利用它来开发插件或完整的工具平台的商业客户来说，Eclipse代表了一种经过验证的、可靠的、可扩展的技术。利用这种技术，人们可以快速设计、开发和部署商业产品。

对于全世界数以千计的学生和研究者来说，Eclipse代表了一个创新、自由和实验的稳定平台。对于所有的这些个人、团队和组织来说，Eclipse是一个用于工具集成的独立于特定厂商的平台。这一平台由一个生机勃勃的Eclipse体系所支持。

独立于特定厂商的Eclipse平台以行业标准为基础。它支持众多的工具、平台和语言。Eclipse技术是无版权费用的，并具有全世界范围内的可重新分发权。该平台是全新建造的，以达到可扩展和提供杰出工具的目标。开发Eclipse基于开源协定。该协定包含开放、透明、基于价值和合作开发等诸多特性。每一个人都能参与其中，并提供添加项。所有的计划都在公开范围内制定。该平台和开源开发组织创建了一个创新、创造性和自由的环境。在当今众多的软件工具环境中，Eclipse是独一无二的。

软件工具行业正在经历从技术商品化到公司合并的巨大转变。新的技术尝试正在重新设计，同时工具基础结构的通用集被不断改进以成为行业标准。成功的开发者和开发模式正面临适应新技术、新的更有效的方法的挑战。旧的商业模式正面临免费软件的挑战，同时，新的商业模式逐渐涌现。

软件工具行业非常感谢Eric Clayberg和Dan Rubel的这本权威书籍。这本书提供了一个知识库。开发人员、工程师和用户均可利用这一知识库来学习和使用Eclipse技术。这样，他们就可以对这些技术和工业变革的原动力做出响应。

Eric和Dan拥有很长的开发软件工具的职业生涯。他们两人均拥有丰富的经验：二十年的Smalltalk开发经验、十三年的Java开发经验和九年的Eclipse开发经验。他们和众多厂商和顾客建立了良好关系，这使得他们可以第一时间使用必要元素以创建成功的软件。他们可以将这种技术的直接经验和用户体验相结合，以写出一本提供创建商业级质量Eclipse扩展项过程的详细描述的书。

本书为新开发者提供了插件开发整个过程的介绍和概述，包括获得高质量效果的所有最佳做法。对于有经验的Eclipse开发人员来说，本书可以作为一本参考书籍。它详细讨论了API，并阐述了众多的示例。同时，本书也为新、老开发人员提供了详细的教程。Eric和Dan添加了他们丰富的用户界面开发经验，也演示了如何使用Eclipse SWT UI。SWT是所有Eclipse UI开发的基石。作者清晰地描述了创建工具软件过程中面临的开发挑战，同时提供了已验证的、深入的解决方案。

如果你是一名开发人员、工程师或想要开发或使用Eclipse的用户，本书提供了基础知识和参考。同时，本书也提供了向开源Eclipse项目提交添加项和开发商业软件的基础知识。

——Skip McGaughey

序 言 二

在20世纪90年代，Java还处在早期阶段。学习Java库，只要学习四五个包中的一些类文件。今天，Java类库已经变得更大更复杂，给现在想要学习Java的开发人员带来了一个巨大的难题。与Java类似，数年时间里，Eclipse平台也逐渐成长。因此学习Eclipse 3.4需要比学习Eclipse以往版本花费更多的时间和精力。Eclipse平台的一条重要的原则就是：一个插件必须与Eclipse平台和其他插件无缝地集成。为了达到无缝集成，插件开发者必须懂得为Eclipse开发软件的最佳实践、约定和策略。本书包含了创建你为之骄傲的Eclipse插件所需要的各个方面。

在开发Favorites插件过程中，详细讨论了Eclipse标准窗口小部件工具集（SWT，Standard Widget Toolkit）和JFace框架。这样你就可以学会如何创建专业外观的用户界面，如视图、编辑器、首选项页和对话框。除了附加的常用主题之外，还详细讨论了诸如用户界面设计等较难理解的Eclipse主题（比如，创建功能部件和产品商标）。此外，本书还给出了我见过的最好的关于使用Ant的讨论，该讨论涉及从一个目标为多个版本的Eclipse的源代码文件创建产品。

在插件声明清单和实现插件的功能行为所必需的Java代码之间存在一种扩展点机制和重要联系，而刚开始使用Eclipse的Java开发者要理解这些内容有一定难度。本书可作为一张使用由Eclipse平台定义的插件开发环境（PDE，Plug-in Development Environment）和扩展点的路线图。本书还提供了以下内容：开发者应当了解的本应在清单中描述的插件的各个方面，如何使用已有扩展点开发插件，以及如何创建其他开发者可以进一步提交添加项的扩展点。

当我初次接触到CodePro，我被两方面吸引住了：它给Eclipse带来的生产效率和它与Eclipse平台结合的紧密程度。在尝试使用CodePro一段时间后，它就已经成为我开发工具集不可或缺的一部分。通过描述他们在开发CodePro中获取的广泛经验，Eric和Dan完成了一项如此优异的工作。这项工作就是本书中所描述的创建高质量和专业外观的Eclipse产品所必需的插件开发的各个方面。

——Simon Archer



前言

1999年末，当Eclipse初次出现在我们面前时，我们被IBM想要解决的问题的重要性触动了。IBM打算将其所有开发环境整合于一个单一代码库之上。当时，IBM正在使用一种技术融合，它融合了C/C++、Java和Smalltalk。

许多最重要的IBM开发工具，实际上均采用Smalltalk语言编写。Smalltalk是一种适用于创建复杂开发工具的编程语言，但它在与诸如Java之类的语言的竞争中，很快便丧失了一部分市场份额。IBM拥有世界上最多的Smalltalk开发者，然而，在IBM外部，Smalltalk并没有获得广泛的行业支持。同时，也几乎没有能够开发基于Smalltalk的各种附加软件的合格独立软件厂商（Independent Software Vendor, ISV）。

同时，Java也提供丰富的应用程序编程接口（Application Programming Interface, API）来用于开发最新的基于Web的企业应用。更重要的是，Java是一种面向对象（Object-oriented, OO）的语言，这就意味着IBM那些高水平的面向对象开发人员可以重新发挥作用。这些为数众多的开发人员是在多年的基于Smalltalk开发各种工具过程中所培养起来的。实际上，IBM VisualAge Smalltalk和VisualAge Java（VisualAge Smalltalk是VisualAge家族的第一个产品，VisualAge Java也使用了该品牌）开发环境是由杰出的国际对象技术组织（Object Technology International, OTI）所开发的。IBM将该组织的主要目标定位于创建一个基于Java的高可扩展性集成开发环境。Eclipse就这样诞生了。

OTI可以利用其高水平的面向对象的开发经验来开发一个具有无比强大功能、适应性和可扩展性的IDE。该组织可以重复利用众多的功能，在过去的二十年里，这些功能使得那些基于Smalltalk的IDE如此之流行。这些功能的重复利用也将IDE（集成开发环境）开发技能向前推进了一个数量级。

Java世界从未见过如此强大、如此引人注意的Eclipse。但它就这样诞生了，和微软的.NET一道成为世界上最好的开发环境。这也使得Eclipse成为开发者努力扩大其作品使用范围的最好的平台。Eclipse是完全免费和开源的，这一事实也是很令人惊喜。这个只要有计算机就能获取的开放的、可扩展的IDE，为潜在的工具开发者提供了强大的推动力。

Eclipse的这些特性对于我们来说也是如此。在Instantiations和早期的ObjectShare，我们花费了十几年的时间关注于为不同的IDE创建附加工具。在那时，我们已经开始为Digital的Smalltalk/V开发附加软件，再转到为IBM的VisualAge Smalltalk，最终为IBM的VisualAge Java（包括曾得奖的VA Assist产品和世界上最早的重构工具之一——jFactor）开发各种工具。但这些插件大多不具有良好的格式，当然，也没有较好的标准化。小范围内的共享（如VisualBasic）和来源广泛的用户自定义库折磨着这些开发环境，也在折磨着我们。

作为IBM的高级商业伙伴，我们幸运地与负责Eclipse开发的IBM公司的众多人员建立了长期的、可信赖的合作关系。这种合作关系意味着我们可以在一种独特的位置审视相应的技术，并几乎在世界上其他人第一次听到Eclipse的一年半之前，我们就已经使用Eclipse进行日常开发。当IBM最终于2001年中向世界发布Eclipse，我们的小组为IBM开发了第一批演示程序中的一部分。在那一年随后的时间里，当IBM发布第一个基于Eclipse的商业开发工具WebSphere Studio Application Developer

v4.0 (v4.0是为了与其当时的VisualAge for Java v4.0保持一致)，我们的CodePro产品也在同一天成为WebSphere（也适用于Eclipse）最早的可获取的商业附加软件。

今天，CodePro产品为Eclipse及基于Eclipse的各种IDE增加了数百项功能。在过去的几年，开发CodePro给我们提供了一个机会来从其他人很少能及的高度来学习开发Eclipse的各种细节（当然除了IBM和OTI的开发者们之外，他们每天都和Eclipse呆在一起）。CodePro也作为许多在这本书中所展示的想法与技术的一个测试温床，这也为我们提供了一个从哪里开始着手撰写本书的独特视角。

本书目标

本书的英文版原标题为《Eclipse: Building Commercial-Quality Plug-ins》，深入描述了为Eclipse和IBM软件开发平台（Software Development Platform, SDP, IBM的商业版本Eclipse开发环境）开发商业级质量标准的扩展组件的过程。对我们而言，商业级质量是商业级标准或高标准同义词。创建一个商业级质量的扩展组件意味着要比集成到Eclipse最小要求做得更多，也意味着要致力于商业软件产品所需达到的“合适与完善”所要求的细节。

在Eclipse插件世界中，很少有人会花多余的时间，因而绝大部分插件都停留在开源、业余水平。对于那些创建高质量插件感兴趣的人们（这些人就是软件公司打算开发基于Eclipse的产品的主要原因）来说，有许多细节要考虑。本书致力于包含插件开发的整个过程，包括所有为达到高质量所必需的附加步骤。本书有几个互补的目标：

- 为新用户提供使用Eclipse的简要介绍
- 为有经验的Eclipse用户想要扩展知识与提高基于Eclipse的产品质量提供参考
- 为新用户和有经验的用户提供创建复杂Eclipse插件的详细教程

前三章介绍了Eclipse开发环境，并列出了创建简单插件的主要步骤。这些章的主要目的是帮助新的Eclipse开发人员很快能创建插件，并验证他们的各种想法。

第1章向读者介绍了开发插件所必需的Eclipse最小工具集。该章主要是对Eclipse IDE和相关工具（这一主题可以独立成书）进行简要介绍。熟练的Eclipse用户可以跳过该章。

第2章介绍了我们整本书中都将要使用的示例，同时简要介绍从开始创建一个可工作的插件的每一个步骤。第3章从较高的角度对Eclipse的基础结构和插件与扩展点的结构进行了介绍。

第4、第5章包含标准窗口小部件集（Standard Widget Toolkit, SWT）和JFace相关内容。这两者是所有Eclipse用户界面（User Interface, UI）的基石。这两章可作为一个单独的参考；主要为用户继续学习提供足够的细节。所有这些主题的内容对于整本书籍的阅读是足够丰富的。

接下来的章节，是本书的主要内容，主要描述了插件开发的各个方面，为读者如何解决将要面对的挑战提供深入的内容。每一章均从问题的不同角度进行阐述。包括概述、细节描述、面临挑战与解决方法的讨论、图表、屏幕截图、规范编程示例、相关API和总结。

本书通过这样的组织结构，以便于插件项目所必需的最重要的相关知识均能在书的上半部分出现。一些有关于打包与创建的素材则放于上半部分靠后的位置（如功能与产品构建）。这种组织模式留下了几个主题。这些主题并非对每一个插件都同样重要，但对于创建商业级质量插件十分重要。这一部分主题在本书的下半部分。这些主题根据重要程度和与前述几章的相关性进行排列。比如，国际化，它并不是关键性的，如果你真正努力去学习，它也不难。然而，它对于本书的前提是十分重要的。所以我们觉得它应该成为一个我们将要讲述的主题。我们假设读者不是一个Eclipse专家（或者甚至不是一个插件开发者），我们试图带领读者浏览重要步骤的尽可能多的细节。也就是说，从某种程度来看，

这一章是介绍性的。这一章也是大部分插件开发者几乎遗忘并具有很少经验的一章。

有时候，开发者需要一个快速解决方案。而在另一些时候，同一个开发者对于开发过程中的某个特定方面需要深入细致的知识。本书写作意图是为读者提供几条掌握并应用知识的方法。这样，这两种要求就都可以得到满足。相关的API包括在一部分章节。因此，本书也可作为开发过程中一本独立的参考手册，这样，读者就不用再在IDE中查找那些API了。API的大部分描述都来源于Eclipse平台的Javadoc。

作为Eclipse的创始者和基于Eclipse的技术的主要使用者，IBM合理地关注新的插件能否满足其一贯坚持的高质量标准。为达成这一目标，IBM建立了一个严格的Rational软件合格认证项目(Ready for Rational Software, RFRS)。这一项目致力于保证高质量Eclipse与IBM软件开发平台的插件的可用性。RFRS认证应当成为所有想要创建与市场化Eclipse插件的开发者的终极目标之一。本书的每一章均包含了RFRS认证相关的范围与策略等内容。

作为各章内容一部分的各种示例描述了创建稳定的Eclipse插件的各个方面。你在学习过程中可以看到这些示例在不断演化。当作为参考手册而不是从头到尾地阅读本书时，你将有可能翻阅至某一章，而实际上，你要找的内容在另外一章。为了减小这种事件发生的概率，每一章都包含许多对其他章相关内容的交叉引用。

目标读者

本书所适合的读者包括：打算开发集成于Eclipse或基于Eclipse的其他产品的软件的Java工具开发者、想要自定义开发环境的高级Eclipse用户或对于是什么让Eclipse工作感兴趣的开发人员。要使用本书，你不必成为一名Eclipse专家，因为我们在第1章中，已经介绍了绝大部分你在使用Eclipse时必须掌握的内容。同时，我们不假设需要任何的Eclipse的预备知识，我们仅仅期望读者是对Java有较好了解的相对熟练的开发者，并对XML有一定的了解。

第3版新增内容

在该版本中，我们使用了和前两个版本一样的Favorites视图示例，但修改了很多内容，并从头开始重写了代码。一些Eclipse概念，如视图和编辑器，是类似的，但具有附加功能；而一些其他内容，如命令、GEF（图形化编辑框架）和PDE（插件开发环境）构建已经添加进来。以下是第3版的一些主要变化：

Eclipse命令框架

Eclipse命令框架代替了旧的操作框架。全书使用旧的操作框架的地方均描述为如何使用新的命令框架完成同样工作的内容所替代。这在第6章尤其明显。在该章中，前半部分几乎全部含有关于新的命令框架。第7章和第8章也包含了许多描写使用视图和编辑器命令的新内容。

Eclipse 3.4和Java 5

本书中所有屏幕截图、文本和代码示例均升级至采用最新的Eclipse 3.4[⊖] API和Java 5语法。第1章几乎完全重写，以包含Eclipse 3.4的新功能，这些新功能包括了使用Mylyn的介绍和对新的首选项的讨论。第1、第2章还包括了Eclipse中PDE和SWT工具的相关介绍。

⊖ 当前最新版本的Eclipse为Eclipse 3.5 Galileo，最新版本的Java SE为JDK 6。本书使用Eclipse 3.4和Java 5。——译者注

新的GEF章节

GEF, 来源于Eclipse.org的图像编辑框架, 提供了用于创建动态交互图形用户界面元素的工具集。第20章带领你一步步地学习创建基于GEF的视图的全过程。该视图用于图形展示收藏夹项及其底层资源的关系。然后, 我们创建一个基于GEF的编辑器。该编辑器具有添加、移动、调整大小和删除表示那些收藏夹项的图形元素的功能。

使用PDE以测试其功能

在过去几年中, PDE构建过程已经逐渐成熟。本书上一版本中的第19章的特殊Ant脚本, 已经使用Eclipse PDE构建过程的指导取代。这些内容将指导自动组装你的产品以用于分发。

新的“p2”更新站点创建描述

“p2”首次出现于Eclipse 3.4, 用于替代较老的升级管理器。我们带领你浏览使用更新站点的过程, 然后再帮你建立你自己的更新站点, 参见18.3节。

致 谢

作者要感谢所有为本书的出版付出过辛勤劳动的人, 同时, 也要感谢那些在本书创作过程中给予我们支持和鼓励的人们。

致我们Instantiations的同事, 你们给了我们时间和鼓励来写作本书, 包括: Rick Abbott、Brent Caldwell、Devon Carew、Jim Christensen、Taylor Corey、Dianne Engles、Marta George、Nick Gilman、Seth Hollyman、Mark Johnson、Ed Klimas、Tina Kvavle、Alexander Lobas、Warren Martin、David Masulis、Nancy McClure、Steve Messick、Alexander Mitin、Gina Nebling、John O'Keefe、Keerti Parthasarathy、Phil Quitslund、Mark Russell、Rob Ryan、Andrey Sablin、Balaji Saireddy、Konstantin Scheglov、Chuck Shawan、Bryan Shepherd、Julie Taylor、Mike Taylor、Solveig Viste、Brian Wilkerson和Jaime Wren。

特别感谢Jaime Wren, 他为第20章做了大量的基础性研究, 并提供了原始内容。

感谢我们经纪人Laura Lewin和Studio B的同事们。你们从第一天起就鼓励我们, 并让我们按照我们的方式较轻松地工作。

感谢我们的编辑Greg Doench和John Neidhart、产品编辑Elizabeth Ryan和Kathleen Caren、文字编辑Marilyn Rash和Camie Goffi、助理编辑Michelle Housley和Mary Kate Murray、美术指导Sandra Schroeder、营销经理Brandon Prebynski和Beth Wickenhiser以及Pearson出版社的同事们, 感谢你们的鼓励和为本书的出版所付出的艰辛劳动。

感谢Simon Archer和Robert Konigsberg, 你们为本书不同版本提供了很多的修改建议, 并帮助我们从不方面改进。

感谢Linda Barney, 帮我们对本书第2版进行修饰和修改。

感谢我们的技术评审, 他们从许多方面帮助我们对本书进行改进, 包括: Matt Lavin、Kevin Hammond、Mark Russell、Keerti Parthasarathy、Jaime Wren、Joe Bowbeer、Brian Wilkerson、Joe Winchester、David Whiteman、Boris Pruesmann、Balaji Saireddy和Raphael Enns。

感谢本书英文版第1、第2版为我们提供勘误的众多读者, 勘误表被本书英文版第3版采纳: Bruce Gruenbaum、Tony Saveski、James Carroll、Tony Weddle、Karen Ploski、Lori Kissell、Brian Vosburgh、Peter Nye、Chris Lott、David Watkins、Simon Archer、Mike Wilkins、Brian Penn、

Bernd Essann、Eric Hein、Dave Hewitson、Frank Parrott、Catherine Suess、William Beebe、Janine Kovack、David Masulis、Jim Norris、Jim Wingard、Roy Johnston、David Karr、Chris Gage、Paul Tamminga、Asim Ullah和Ebru Aktuna。

感谢本书英文版系列丛书的编辑Erich Gamma、Lee Nackman和John Weigand。他们提供了许多体贴的建议。正是他们的不断努力，使得Eclipse成为世界上最好的开发环境。

我们还必须感谢我们的妻子，Karen和Kathy。她们展现了极大的耐心。以及我们的孩子们，Beth、Lauren、Lee和David。他们是我们力量的源泉。

作者简介



Eric Clayberg是Instantiations公司产品开发部高级副总裁。Eric是一位具有丰富经验的软件技术专家、产品开发人员、企业家和具有超过17年商业软件开发经验的项目经理。这17年开发经验包含了12年的Java开发经验和9年的Eclipse开发经验。他是十几个商业Java和Smalltalk附加产品的主要作者和架构者，其中包括了广为流行的WindowBuilder Pro、CodePro和获奖的VA Assist产品系列。他拥有麻省理工学院的理学学士学位、哈佛大学的MBA学位。是两家成功的软件公司ObjectShare和Instantiations的共同创始人之一。



Dan Rubel是Instantiations公司的首席技术执行官。他是一名成功的商人，也是面向对象技术方面的设计与应用的专家。他具有15年以上的商业软件开发经验，其中包括了13年的Java开发经验和9年的Eclipse开发经验。他是几个成功的商业产品的架构师和项目经理。这些商业产品包括RCP Developer、WindowTester、jFactor和jKit。他在其他几个商业产品中扮演关键设计与领导角色。这些商业产品包括VA Assist和CodePro。他拥有Bucknell的理学学士学位，是Instantiations公司的共同创始人之一。

Instantiations是IBM高级商业伙伴，为Eclipse和IBM的VisualAge、WebSphere和Rational产品系列开发了许多商业附加软件。Instantiations公司是Eclipse基金会成员，是Eclipse开源组织的主要贡献者之一，负责Eclipse Collaboration Tools项目（即Koi项目）和Eclipse Pollinate项目（即Beehive项目）。

如何联系我们

我们做了许多努力以保证本书内容是及时的、准确的，然而Eclipse的发展是十分迅速的。你可能会发现书中展示的内容与你在使用Eclipse时有所区别。Eclipse的UI（用户界面）这些年来在不断发展。最新的3.4版本和即将发布的3.5版本也是如此。我们专注于Eclipse 3.4，并用其开发了所有示例。本书完成于Eclipse 3.4完成之后Eclipse 3.5早期阶段。如果你使用的是较老的或更新的Eclipse版本，你将会接触到不同视图、对话框和与本书截图有些不同的向导。

- 本书技术内容相关问题，请发邮件至：
info@qualityeclipse.com
- 本书相关项目的源代码可以在以下地址找到：
www.qualityeclipse.com/projects
- 本书英文版的勘误表可以在以下地址找到：
www.qualityeclipse.com/errata
- 本书所使用和描述的工具可以在以下地址找到：
www.qualityeclipse.com/tools

目 录

读者赞誉	
译者序	
序言一	
序言二	
前 言	
第1章 使用Eclipse工具	1
1.1 起步	1
1.1.1 获取Eclipse	1
1.1.2 安装	2
1.2 Eclipse工作台	2
1.2.1 透视图、视图和编辑器	4
1.2.2 操作	7
1.3 设置Eclipse	9
1.3.1 工作台首选项	9
1.3.2 Java首选项	10
1.3.3 导入与导出首选项	11
1.4 创建项目	11
1.4.1 使用新建Java项目向导	12
1.4.2 .classpath和.project文件	13
1.4.3 使用Java包向导	14
1.4.4 使用Java类向导	14
1.5 导航	15
1.5.1 打开类型对话框	15
1.5.2 类型层次结构视图	16
1.5.3 转至行	16
1.5.4 大纲视图	16
1.5.5 快速访问	17
1.6 搜索	17
1.6.1 文件搜索	17
1.6.2 Java搜索	18
1.6.3 其他搜索菜单选项	19
1.6.4 工作集	19
1.7 编写代码	20
1.7.1 Java编辑器	20
1.7.2 模板	24
1.7.3 重构	25
1.7.4 本地历史记录	26
1.7.5 文件扩展名关联	28
1.8 使用CVS进行团队开发	29
1.8.1 开始使用CVS	30
1.8.2 从CVS中导出项目	30
1.8.3 与库同步	31
1.8.4 比较与替代资源	31
1.8.5 CVS标签装饰器	32
1.9 运行程序	33
1.9.1 启动Java程序	33
1.9.2 启动配置	34
1.10 调试简介	35
1.10.1 设置断点	35
1.10.2 使用调试视图	36
1.10.3 使用变量视图	36
1.10.4 使用表达式视图	36
1.11 测试简介	37
1.11.1 创建测试用例	37
1.11.2 运行测试用例	37
1.12 Mylyn简介	38
1.13 总结	41
参考文献	41
第2章 简单插件示例	42
2.1 收藏夹插件	42
2.2 创建插件项目	42
2.2.1 新建插件项目向导	42
2.2.2 定义插件	42

2.2.3 定义视图	43	3.3.3 插件依赖项	71
2.3 评审生成代码	44	3.3.4 扩展项与扩展点	73
2.3.1 插件清单	44	3.4 启动器或插件类	74
2.3.2 启动器或插件类	48	3.4.1 启动与关闭	74
2.3.3 收藏夹视图	50	3.4.2 插件早期启动	74
2.4 构建产品	52	3.4.3 静态插件资源	74
2.4.1 手动构建	52	3.4.4 插件首选项	75
2.4.2 使用Apache Ant构建	53	3.4.5 插件配置文件	75
2.5 安装并运行产品	56	3.4.6 插件与AbstractUIPlugin	77
2.6 调试产品	56	3.5 插件模型	77
2.6.1 创建配置文件	56	3.5.1 平台	78
2.6.2 选择插件和片段	57	3.5.2 插件与包	78
2.6.3 启动运行时工作台	58	3.5.3 插件扩展项注册表	79
2.7 PDE视图	58	3.6 日志	79
2.7.1 插件注册表视图	58	3.6.1 状态对象	80
2.7.2 插件视图	58	3.6.2 错误日志视图	80
2.7.3 插件依赖项视图	59	3.6.3 处理错误(与其他状态)	81
2.7.4 插件手动搜索	59	3.7 Eclipse插件	81
2.7.5 插件探测器	59	3.8 总结	82
2.8 编写插件测试	60	参考文献	82
2.8.1 测试准备	60	第4章 标准窗口小部件工具集	83
2.8.2 创建插件测试项目	60	4.1 SWT历史与目标	83
2.8.3 创建插件测试	60	4.2 SWT窗口小部件	85
2.8.4 运行插件测试	63	4.2.1 简单独立示例	85
2.8.5 卸载收藏夹插件	64	4.2.2 窗口小部件生命周期	87
2.9 本书示例	64	4.2.3 窗口小部件事件	87
2.10 总结	65	4.2.4 抽象窗口小部件类	88
参考文献	65	4.2.5 最高级类	90
第3章 Eclipse基础结构	66	4.2.6 常用窗口小部件	92
3.1 结构概述	66	4.2.7 菜单	108
3.1.1 插件结构	67	4.2.8 其他窗口小部件	110
3.1.2 工作区	67	4.3 布局管理	110
3.2 插件目录与JAR文件	68	4.3.1 填充布局(FillLayout)	110
3.2.1 链接文件	68	4.3.2 行布局(RowLayout)	111
3.2.2 混合途径	69	4.3.3 网格布局(GridLayout)	113
3.3 插件清单	69	4.3.4 表单布局(FormLayout)	115
3.3.1 插件声明	70	4.4 资源管理	117
3.3.2 插件运行时	71	4.4.1 颜色	117

4.4.2 字体	117	6.6.2 菜单中的组	154
4.4.3 图像	117	6.6.3 定义菜单项和工具栏按钮	154
4.5 GUI构建器 (GUI Builder)	118	6.6.4 操作的图像	155
4.6 总结	119	6.6.5 插入点	155
参考文献	119	6.6.6 创建操作代表	156
第5章 JFace查看器	121	6.6.7 手动测试新建操作	157
5.1 面向列表的查看器	121	6.6.8 为新操作添加测试	158
5.1.1 标签提供者	122	6.6.9 讨论	160
5.1.2 内容提供者	122	6.7 对象操作	160
5.1.3 查看器排序器	123	6.7.1 定义基于对象的操作	161
5.1.4 查看器过滤器	123	6.7.2 操作过滤与可用	163
5.1.5 StructuredViewer类	123	6.7.3 IObjectActionDelegate	167
5.1.6 ListViewer类	125	6.7.4 创建基于对象的子菜单	167
5.1.7 TableViewer类	127	6.7.5 手动测试新操作	168
5.1.8 TreeViewer类	130	6.7.6 为新操作添加测试	168
5.2 文本查看器	132	6.8 视图操作	169
5.3 总结	134	6.8.1 定义视图上下文子菜单	169
参考文献	134	6.8.2 定义视图上下文菜单操作	170
第6章 命令与操作	135	6.8.3 IViewActionDelegate	171
6.1 命令	135	6.8.4 定义视图工具栏操作	171
6.2 菜单和工具栏添加项	138	6.8.5 定义视图下拉子菜单和操作	172
6.2.1 定义最高级菜单	138	6.8.6 手动测试新操作	172
6.2.2 添加至已有最高级菜单	138	6.8.7 为新操作添加测试	172
6.2.3 定义最高级工具栏项	139	6.8.8 视图上下文菜单标识符	172
6.2.4 限制最高级菜单与工具栏项的 可见性	139	6.9 编辑器操作	174
6.2.5 定义基于选择的上下文菜单项	140	6.9.1 定义编辑器上下文菜单	174
6.2.6 定义视图相关菜单或工具栏项	143	6.9.2 定义编辑器上下文操作	175
6.2.7 定义编辑器相关的菜单或工具栏 项目	143	6.9.3 IEditorActionDelegate	175
6.2.8 动态菜单添加项	144	6.9.4 定义编辑器最高级菜单	176
6.2.9 locationURI	144	6.9.5 定义编辑器最高级操作	176
6.2.10 visibleWhen表达式	145	6.9.6 定义编辑器工具栏操作	177
6.3 处理器	148	6.9.7 为新操作添加测试	177
6.4 键绑定	150	6.9.8 编辑器上下文菜单标识符	177
6.5 IAction与IActionDelegate	151	6.10 操作和键绑定	178
6.6 工作台窗口操作	152	6.10.1 将命令与操作相关联	178
6.6.1 定义工作台窗口菜单	153	6.10.2 键盘可访问性	179
		6.11 RFRS相关事项	180
		6.12 总结	180

参考文献	180	7.9.2 视图立即保存 (RFRS 3.5.16)	227
第7章 视图	181	7.9.3 视图初始化 (RFRS 3.5.17)	227
7.1 视图声明	182	7.9.4 视图全局操作 (RFRS 3.5.18)	227
7.1.1 声明视图类别	182	7.9.5 保存视图状态 (RFRS 3.5.19)	228
7.1.2 声明视图	183	7.9.6 注册上下文菜单 (RFRS 5.3.5.8)	228
7.2 视图部件	184	7.9.7 视图操作过滤程序 (RFRS 5.3.5.9)	228
7.2.1 视图方法	184	7.10 总结	229
7.2.2 视图控件	184	参考文献	229
7.2.3 视图模型	185	第8章 编辑器	230
7.2.4 内容提供者	194	8.1 编辑器声明	231
7.2.5 标签提供者	195	8.2 编辑器组件	233
7.2.6 查看器排序器	196	8.2.1 编辑器方法	233
7.2.7 查看器过滤器	198	8.2.2 编辑器控件	234
7.2.8 视图选择	199	8.2.3 编辑器模型	236
7.2.9 实现propertyTester	199	8.2.4 内容提供者	242
7.3 视图命令	199	8.2.5 标签提供者	243
7.3.1 模型命令处理器	200	8.3 编辑	244
7.3.2 上下文菜单	200	8.3.1 单元格编辑器	244
7.3.3 工具栏按钮	203	8.3.2 变更监听器	246
7.3.4 下拉菜单	204	8.3.3 单元格验证器	247
7.3.5 键盘命令	205	8.3.4 编辑与选择	249
7.3.6 全局命令	205	8.4 编辑器生命周期	249
7.3.7 剪贴板命令	206	8.4.1 修改过的编辑器	249
7.3.8 拖放支持	210	8.4.2 切换页面	250
7.3.9 内联编辑	215	8.4.3 保存内容	251
7.4 链接视图	218	8.5 编辑器命令	251
7.4.1 选择提供者	218	8.5.1 上下文菜单	251
7.4.2 可适配对象	218	8.5.2 编辑器添加程序	253
7.4.3 选择监听器	218	8.5.3 编辑器命令而不是编辑器添加 程序	257
7.4.4 打开编辑器	219	8.5.4 撤销/重做	259
7.5 保存视图状态	220	8.5.5 剪贴板操作	266
7.5.1 保存本地视图信息	220	8.6 链接编辑器	266
7.5.2 保存全局视图信息	222	8.7 RFRS相关事项	266
7.6 测试	225	8.7.1 使用编辑器进行编辑或浏览 (RFRS 3.5.9)	266
7.7 图像缓存	225	8.7.2 编辑器生命周期 (RFRS 3.5.10)	266
7.8 自动调整大小的表列	226		
7.9 RFRS相关事项	227		
7.9.1 用于导航的视图 (RFRS 3.5.15)	227		

8.7.3 访问全局操作 (RFRS 3.5.11) ...	267	10.2.1 添加视图和占位符	285
8.7.4 当对象被删除时关闭 (RFRS 3.5.12)	267	10.2.2 添加快捷方式	287
8.7.5 同步外部更改 (RFRS 3.5.14) ...	268	10.2.3 添加操作集	288
8.7.6 注册编辑器菜单 (RFRS 5.3.5.2)	268	10.3 RFRS相关事项	289
8.7.7 编辑器操作过滤器 (RFRS 5.3.5.3)	268	10.3.1 为长时间任务创建 (RFRS 5.3.5.10)	289
8.7.8 未保存的编辑器更改 (RFRS 5.3.5.4)	268	10.3.2 扩展已有透视图 (RFRS 5.3.5.11)	289
8.7.9 为更改过的资源添加前缀 (RFRS 5.3.5.5)	268	10.3.3 添加操作至窗口菜单 (RFRS 5.3.5.15)	289
8.7.10 编辑器大纲视图 (RFRS 5.3.5.6)	268	10.4 总结	290
8.7.11 与大纲视图同步 (RFRS 5.3.5.7)	269	参考文献	290
8.8 总结	269	第11章 对话框与向导	291
参考文献	269	11.1 对话框	291
第9章 资源更改跟踪	270	11.1.1 SWT对话框与JFace对话框	291
9.1 IResourceChangeListener	270	11.1.2 普通SWT对话框	291
9.1.1 IResourceChangeEvent	270	11.1.3 普通JFace对话框	292
9.1.2 IResourceDelta	271	11.1.4 创建JFace对话框	293
9.2 处理更改事件	272	11.1.5 对话框单元	295
9.3 批处理更改事件	274	11.1.6 对话框的初始位置和大小	295
9.4 进度监视器	276	11.1.7 可调整大小的对话框	295
9.4.1 IProgressMonitor	276	11.1.8 收藏夹视图过滤器对话框	296
9.4.2 用于显示进度的类	276	11.1.9 细节对话框	299
9.4.3 工作台窗口状态栏	278	11.1.10 打开对话框——查找父shell ...	305
9.4.4 IProgressService	279	11.2 向导	306
9.5 被延迟的更改事件	279	11.2.1 IWizard	307
9.6 总结	280	11.2.2 IWizardPage	308
参考文献	280	11.2.3 IWizardContainer	309
第10章 透视图	281	11.2.4 嵌套的向导	310
10.1 创建透视图	281	11.2.5 启动向导	310
10.1.1 透视图扩展点	282	11.2.6 向导示例	312
10.1.2 透视图工厂	282	11.2.7 对话框设置	314
10.1.3 IPageLayout	284	11.2.8 基于选择的页面内容	314
10.2 改进已有透视图	285	11.2.9 基于前一页面的页面内容	318
		11.3 RFRS相关事项	320
		11.3.1 向导外观 (RFRS 3.5.2)	320
		11.3.2 在编辑器中打开新文件 (RFRS 3.5.6)	320

11.3.3 新项目切换透视图 (RFRS 3.5.7)	321	13.3.1 属性视图API	346
11.3.4 显示新对象 (RFRS 3.5.8)	321	13.3.2 属性视图中的收藏夹属性	348
11.3.5 单一页面向导按钮 (RFRS 5.3.5.13)	321	13.4 属性页作为首选项页重用	349
11.4 总结	321	13.5 RFRS相关事项	350
参考文献	321	13.6 总结	351
第12章 首选项页	322	参考文献	351
12.1 创建首选项页	322	第14章 构建器、标记和性质	352
12.2 首选项页API	323	14.1 构建器	353
12.2.1 FieldEditorPreferencePage	324	14.1.1 声明构建器	353
12.2.2 字段编辑器	324	14.1.2 IncrementalProjectBuilder	355
12.2.3 PreferencePage	327	14.1.3 派生资源	360
12.2.4 收藏夹首选项页	327	14.1.4 关联构建器与项目	360
12.2.5 合法性验证	328	14.1.5 触发构建器	362
12.2.6 嵌套首选项页	329	14.2 标记	363
12.2.7 选项卡式首选项页	330	14.2.1 标记类型	363
12.3 首选项API	331	14.2.2 创建并删除标记	365
12.3.1 默认首选项	331	14.2.3 标记属性	366
12.3.2 访问首选项	332	14.2.4 标记解析——快速修复	368
12.3.3 在程序中指定默认值	333	14.2.5 查找标记	371
12.3.4 在文件中指定默认值	334	14.3 性质	372
12.3.5 关联收藏夹视图	334	14.3.1 声明性质	372
12.3.6 监听首选项更改	335	14.3.2 关联构建器与性质	373
12.4 RFRS相关事项	335	14.3.3 IProjectNature	374
12.5 总结	336	14.3.4 必需的性质	375
参考文献	336	14.3.5 冲突的性质	375
第13章 属性	337	14.3.6 性质图像	376
13.1 创建属性	337	14.3.7 关联性质与项目	376
13.1.1 FavoriteItem属性	337	14.4 RFRS相关事项	
13.1.2 资源属性	339	14.4.1 使用构建器以转换资源 (RFRS 3.8.1)	380
13.2 在属性对话框中显示属性	339	14.4.2 不要替代已有构建器 (RFRS 3.8.3)	380
13.2.1 声明属性页	340	14.4.3 不要滥用术语“构建” (RFRS 5.3.8.1)	380
13.2.2 创建资源属性页	342	14.4.4 标记已创建的资源为“派生的” (RFRS 5.3.8.2)	381
13.2.3 创建收藏夹项资源页	343	14.4.5 响应清理构建请求 (RFRS 5.3.8.3)	381
13.2.4 打开属性对话框	345		
13.2.5 IColorProvider	345		
13.3 在属性视图中显示属性	346		

14.4.6 在可能时使用IResourceProxy (RFRS 5.3.8.4)	381	15.6.6 附加文档的使用 (RFRS 5.3.7.5)	405
14.4.7 构建器必须由性质添加 (RFRS 5.3.8.5)	382	15.6.7 提供任务流的概述 (RFRS 5.3.5.34)	405
14.5 总结	382	15.6.8 仅说明一个任务 (RFRS 5.3.5.35)	406
参考文献	382	15.6.9 为每一个步骤提供帮助链接 (RFRS 5.3.5.36)	406
第15章 实现帮助	383	15.7 总结	406
15.1 使用帮助	383	参考文献	406
15.2 实现帮助	385	第16章 国际化	407
15.2.1 新建帮助项目	385	16.1 外部化插件清单	407
15.2.2 插件清单文件	387	16.2 外部化插件字符串	409
15.2.3 内容表 (toc) 文件	388	16.3 使用片段	414
15.2.4 创建HTML内容	390	16.3.1 新建片段项目向导	414
15.3 上下文相关的帮助 (F1)	391	16.3.2 片段清单文件	415
15.3.1 关联上下文ID与项	392	16.3.3 片段项目内容	417
15.3.2 IWorkbenchHelpSystem API	393	16.4 手动测试	417
15.3.3 创建上下文相关的帮助内容	393	16.5 总结	417
15.3.4 上下文扩展点	394	参考文献	417
15.3.5 标记帮助	396	第17章 创建新扩展点	419
15.4 从程序中访问帮助	396	17.1 扩展点机制	419
15.4.1 打开指定帮助页	396	17.2 定义扩展点	419
15.4.2 打开网页	397	17.2.1 创建扩展点	420
15.5 备忘单	398	17.2.2 创建扩展点模式	421
15.5.1 使用备忘单	398	17.2.3 扩展点元素和属性	422
15.5.2 创建简单备忘单	398	17.2.4 扩展点元素语法	425
15.5.3 注册备忘单	400	17.3 扩展点的后台代码	426
15.5.4 添加备忘单命令	402	17.3.1 分析扩展项信息	426
15.5.5 添加命令参数	403	17.3.2 创建代理	427
15.6 RFRS相关事项	404	17.3.3 创建可执行扩展项	429
15.6.1 通过帮助系统提供帮助 (RFRS 3.7.2)	404	17.3.4 清理	431
15.6.2 通过帮助系统提供所有帮助 (RFRS 5.3.7.1)	404	17.4 扩展点文档	431
15.6.3 使用F1激活上下文帮助 (RFRS 5.3.7.2)	405	17.5 使用扩展点	432
15.6.4 实现活动帮助 (RFRS 5.3.7.3)	405	17.6 RFRS相关事项	434
15.6.5 独立帮助的使用 (RFRS 5.3.7.4)	405	17.6.1 文档扩展点 (RFRS 3.10.5)	434
		17.6.2 记录错误 (RFRS 5.3.10.1)	434
		17.7 总结	434

参考文献	434	19.2 使用PDE构建	467
第18章 功能部件、品牌化和更新	435	19.2.1 PDE构建概述	467
18.1 功能部件项目	436	19.2.2 PDE构建过程中的步骤	468
18.1.1 创建新功能部件项目	436	19.2.3 PDE构建过程中的目录	469
18.1.2 功能部件清单文件	437	19.2.4 PDE脚本和模板	469
18.1.3 功能部件清单编辑器	438	19.2.5 创建PDE构建	470
18.1.4 测试功能部件	442	19.2.6 指定编译级别	471
18.2 品牌化	442	19.2.7 运行PDE构建	471
18.2.1 about.html文件	443	19.2.8 自动生成版本限定符	472
18.2.2 about.ini文件	443	19.2.9 保持版本同步	473
18.2.3 产品品牌化	445	19.2.10 构建属性	473
18.3 更新站点	446	19.2.11 自定义PDE目标	475
18.3.1 创建更新站点项目	447	19.2.12 使用不同版本的Eclipse编辑	476
18.3.2 site.xml文件	447	19.3 调试PDE构建过程	477
18.3.3 更新网站	450	19.3.1 自动生成的构建脚本	477
18.3.4 回到功能部件清单	450	19.3.2 使用调试器	478
18.3.5 访问更新站点	451	19.4 总结	478
18.4 RFRS相关事项	453	参考文献	479
18.4.1 不要覆盖产品品牌 (RFRS 3.1.8)	453	第20章 GEF: 图形编辑框架	480
18.4.2 具有品牌的功能部件可见性 (RFRS 5.3.1.9)	453	20.1 GEF体系结构	480
18.4.3 包含添加项信息 (RFRS 5.3.1.10)	453	20.2 GEF模型	481
18.4.4 about.html文件内容 (RFRS 5.3.1.11)	453	20.3 GEF控制器	481
18.4.5 启动画面限制 (RFRS 5.3.1.12)	454	20.3.1 EditPart类	481
18.5 总结	454	20.3.2 最高级EditPart	483
参考文献	454	20.3.3 子EditParts	484
第19章 构建产品	455	20.3.4 连接EditParts	485
19.1 Ant的简要介绍	455	20.3.5 EditPartFactory	488
19.1.1 构建项目	455	20.4 GEF图案	488
19.1.2 构建目标	455	20.4.1 IFigure	489
19.1.3 构建任务	456	20.4.2 Graphics	489
19.1.4 构建属性	458	20.4.3 复杂图案	491
19.1.5 <antcall>任务	463	20.4.4 连接图案	494
19.1.6 macrodef	465	20.4.5 LayoutManager	494
19.1.7 Ant扩展项	466	20.5 Eclipse视图中的GEF	496
		20.6 Eclipse编辑器中的GEF	498
		20.6.1 编辑器输入	499
		20.6.2 回到FavoritesManagerEditPart	500
		20.6.3 绘画编辑器类	500

20.6.4 FavoritesGEFEditor	501	21.4.1 IWorkbenchBrowserSupport	521
20.6.5 用户与GEF的交互	503	21.4.2 LaunchURL	521
20.6.6 编辑菜单	506	21.4.3 OpenEmailAction	522
20.6.7 FreeformLayer和FreeformLayout	508	21.5 扩展点中指定的类型	525
20.6.8 z顺序	509	21.5.1 参数化的类型	525
20.6.9 删除模型对象	511	21.5.2 在不同的插件中引用类	526
20.7 选项板	512	21.6 修改Eclipse以查找部分标识符	527
20.7.1 创建GEF选项板	512	21.6.1 修改Eclipse基础	527
20.7.2 CreateCommand	513	21.6.2 创建全局操作	528
20.8 总结	514	21.6.3 测试新功能	530
参考文献	514	21.6.4 提交更改至Eclipse	530
第21章 高级话题	515	21.7 标签修饰符	530
21.1 高级搜索——引用项目	515	21.7.1 声明标签装饰符	531
21.2 访问内部代码	516	21.7.2 ILightweightLabelDecorator	531
21.2.1 Eclipse新闻组	516	21.7.3 装饰性标签装饰符	533
21.2.2 Bugzilla——Eclipse bug跟踪 系统	516	21.7.4 IDecoratorManager	534
21.2.3 用于访问内部代码的选项	517	21.8 后台任务——Jobs API	535
21.2.4 Eclipse的不同之处	517	21.9 插件ClassLoader	537
21.2.5 相关插件	517	21.10 早期启动	541
21.2.6 使用片段	518	21.10.1 管理早期启动	541
21.3 适配器	518	21.10.2 取消早期启动	542
21.3.1 IAdaptable	518	21.11 富客户端平台	542
21.3.2 使用适配器	518	21.12 总结	542
21.3.3 适配器工厂	519	参考文献	542
21.3.4 IWorkbenchAdapter	520	附录A Eclipse插件和资源	544
21.4 打开浏览器或创建E-mail	521	附录B Ready for Rational Software	552



第1章 使用Eclipse工具

本章讨论了如何使用Eclipse开发环境创建Java应用程序，更重要的是为Eclipse本身创建改进项。我们首先从获得Eclipse和安装它开始介绍。然后，我们对Eclipse UI做了简要介绍，并说明如何对其进行自定义设置。然后，本章介绍了一些重要的Eclipse工具，并描述了如何使用这些工具创建初始Java项目。Eclipse开发人员大多数情况下都是作为开发团队的一名成员，并需要与团队其他成员共享代码。因此，本章也包含如何安装和使用作为Eclipse一部分的并发版本系统（Concurrent Versions System, CVS）相关内容。在创建一个简单的Java项目和类之后，我们将开展有关运行、调试和测试代码的细节讨论。

1.1 起步

在使用Eclipse之前，我们要先从网络下载它，安装并进行一定设置。

1.1.1 获取Eclipse

Eclipse的主网站是www.eclipse.org（图1-1）。在该主页中，你可以看到最新的Eclipse新闻和大量指向在线资源的链接，包括文章、新闻组、bug跟踪（参见21.2.2节）和邮件组。

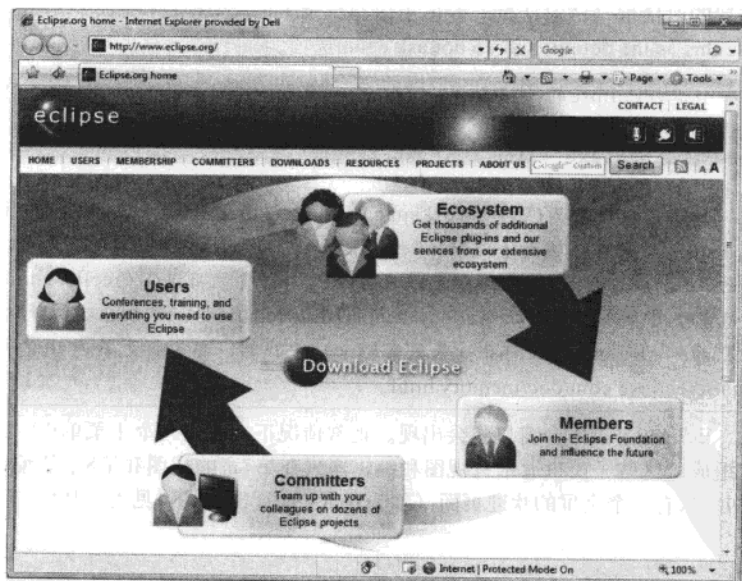


图1-1 Eclipse.org主页面

最新版本的Eclipse可以从主下载页面www.eclipse.org/download下载。从Eclipse 3.3开始,可以获得不同版本的Eclipse,包括适用于Java和J2EE应用开发的版本和适用于RCP和插件开发的版本。如果你想专注于开发插件,请选择下载Eclipse for RCP/Plug-in Developers版本。选择对应于你的平台的链接,保存Eclipse Zip文件至计算机的硬盘上。一般地,Eclipse Zip文件是比较大的(大于150MB),所以,如果你的带宽比较小,就请耐心等待吧。

如果你需要获取其他构建版本(包括其他发布版本和较新的nightly和integration版本),在<http://download.eclipse.org/eclipse/downloads>会提供更详细的下载页面。除非你是在开发Eclipse本身,否则请不要下载integration或nightly版本。不同发布版本的页面均会包含该版本与不同平台版本链接的相关信息。Eclipse支持众多的平台,包括Windows、Linux、Solaris、HP、Mac OS X等。

Java 运行环境 Eclipse是一个Java程序,但它并不包含运行它所必需的Java运行环境(Java Runtime Environment, JRE)。Eclipse 3.4可运行于JRE 1.4之后的所有JRE。绝大部分Java开发者都已经在计算机上安装了相应的JRE。如果你的计算机上没有安装JRE,你可以从java.sun.com下载并安装。

1.1.2 安装

当Eclipse Zip文件成功下载后,将它解压缩至硬盘。Eclipse不会修改Windows注册表,所以它的安装位置是可变的。本书假设它安装至C:\eclipse。

1.2 Eclipse工作台

双击C:\eclipse目录中的eclipse.exe文件以启动Eclipse。Eclipse初次运行时,将会出现一个让你选择工作目录位置的对话框(该对话框一般位于用户目录下面)。为了不每次启动Eclipse都出现该对话框,选中Use this as the default and do not ask again选项。

提示 创建一个运行Eclipse的快捷方式,可提供一种选择不同工作目录和增加Eclipse使用内存的方法。比如:

```
C:\eclipse\eclipse.exe -data C:\MyWorkspace -vmargs -Xms128M -Xmx512M  
-XX:MaxPermSize=256m
```

在该示例中,工作目录设置为C:\MyWorkspace,初始内存设置为128MB,最大内存设置为512MB,最大内存永久保存区域大小(MaxPermSize)设置为256MB。设置工作目录对于在将来与更新版本的Eclipse合并很有帮助。你可以在位于Eclipse安装目录下的eclipse.ini文件中设置这些参数。一份完整的命令行参数的列表(如-vm和-showlocation),可以在联机帮助Workbench User Guide > Tasks > Running Eclipse(参见第15章)中找到。想了解更多内存使用相关信息,请访问www.qualityeclipse.com/doc/memory.html。

稍等一会儿,Eclipse主工作台窗口将会出现。正常情况下,它由一个主菜单栏、工具栏和一些带标题的子窗口组成。这些子窗口通常有视图和编辑器(在第7章的视图和第8章的编辑器中有更详细的介绍)。最初,只有一个全屏的欢迎页面(称为Welcome视图)是可见的,且填充了整个工作台窗口。

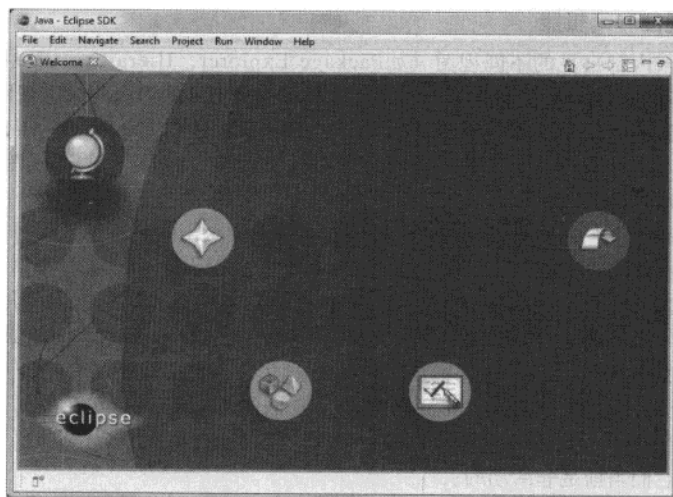


图1-2 Eclipse工作台窗口

Welcome视图在Eclipse第一次启动时会自动打开（它在任何时候均可以通过Help > Welcome打开）。请花费几分钟细致观察它。你会发现它提供了到其他工具和资源的链接。这些链接可以让你开始使用Eclipse。这些链接包括概述、教程和一个简单应用程序列表。

关闭Welcome视图（通过点击标题选项卡中的“X”按钮）后，将会出现几个不同的视图（参见表1-3）。

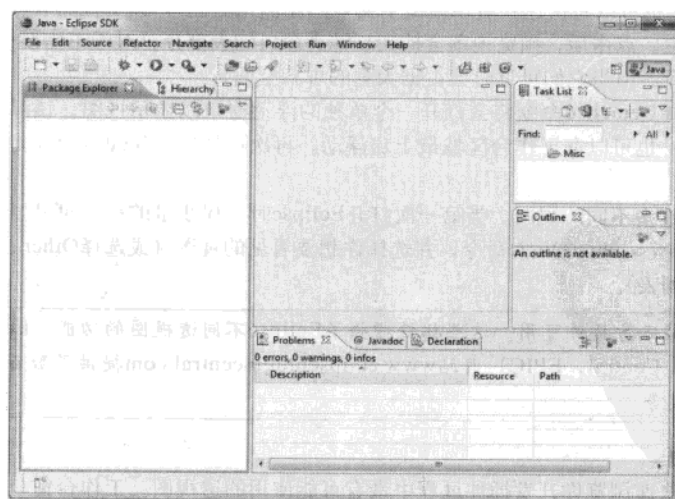


图1-3 当Package Explorer视图处于活动状态的工作台窗口

1.2.1 透视图、视图和编辑器

在Eclipse工作台内部可见的不同视图（如Package Explorer、Hierarchy、Task List、Outline、Problems、Javadoc和Declaration）和编辑器（用于处理不同资源）的组合叫做透视图（perspective）。一个透视图可以看做是Eclipse工作台中的一个页面。不同透视图可以同时打开，每一个在工作台的右上角的透视图工具栏中呈现一个图标（可能包含描述文本）。当前活动的透视图在窗口的标题栏显示其名称，并显示其图标。

视图一般用于浏览资源和修改资源的属性。所有在视图中做出的更改都是即时保存的。相对地，编辑器用来查看或设置特定资源，且遵循打开-保存-关闭（open-save-close）模型。

每一个透视图均具有自己的视图集，但打开的编辑器会被所有打开的透视图共享。任意视图在一个指定的透视图集中只能打开一个实例，然而可以同时打开任意数量相同类型的编辑器。

当前活动视图或编辑器会将其标题栏设置为高亮。并且，它会接受所有的全局动作，如剪切、复制或粘贴。其他所有的子窗口都是非活动的，且它们的标题栏都是灰色的。比如，当你点击Package Explorer视图时，它的标题栏变成高亮，表示它是活动的（图1-3）。同时，其他视图的标题栏变成灰色，表示它们当前是非活动的。

视图和编辑器可以通过拖曳形状边界来改变其大小。形状边界位于该窗口的四周。由于Eclipse以平铺的方式显示它的子窗口，因此将一个子窗口调大意味着将其他窗口调小，反之亦然。

子窗口（pane）可以通过拖曳它们各自的标题栏移动。如果你将一个视图拖至另一个视图上时，这两个视图会重叠显示，通过选项卡区分。选择一个选项卡将使该视图位于栈的顶端。如果一个视图落入其他视图之间，该视图会占据一定的区域，并插入到之前占据该区域的视图的旁边。而之前占据该区域的视图则会缩小其面积以容纳新的视图。

右键点击视图的选项卡，并选择Fast View命令将会使视图停靠至窗口底部边缘的快速查看栏（Fast View）上（你也可以将快速查看栏拖曳至窗口的左侧或右侧）。预览视图会一直停靠在快速查看栏上，除非点击它。点击后，预览视图会扩大，铺满窗口的大部分。对于不需要一直呈现的视图，预览视图是理想选择，但当它们可见时会占据大量的屏幕空间。

右键点击并选择Detached命令将会打开一个单独的浮动窗口以显示视图。该浮动窗口可以移动至工作台区域之外，也可以在工作台区域最上层浮动。再次选择Detached命令则将视图重新返回至工作台窗口中。

Eclipse定义了许多不同的视图。当第一次打开Eclipse时，仅少量的视图可见。若想向透视图添加视图，选择Window > Show View命令，并选择你想要看见的视图（或选择Other...命令以查看系统定义的所有视图的列表）。

提示 有许多第三方插件可用。这些插件增强了Eclipse不同透视图的功能。Eclipse插件中心（Eclipse Plugin Central, EPIC）网站www.eclipseplugincentral.com提供了数百个商业和开源Eclipse插件和服务。

1. Java透视图

此时，我们将快速浏览你开发插件过程中最有可能使用的透视图。工作台窗口初次显示的透视图是Java透视图（图1-3）。

Eclipse包含两个用于开发Java代码的视图。选择Window > Open Perspective > Java命令将打开第一个透视图，即Java透视图。

Java透视图中的主视图是Package Explorer。Package Explorer提供了Java项目中的Java文件和资源文件的层次结构，该层次结构以Java而不是文件为主题排列。

比如，与Navigator视图将Java包显示为嵌套文件夹（参见1.2.1节）不同的是，Package Explorer将每一个包平铺显示为一个单独的元素。项目引用的所有JAR文件也可以通过这种方式浏览。

第二个主要的Java透视图是Java Browsing透视图。选择Window > Open Perspective > Java Browsing来打开该透视图（图1-5）。

Java Browsing透视图包含了一系列的链接视图。这些视图让人想起了在不同的Smalltalk IDE和IBM VisualAge for Java IDE中的浏览器。第一个视图显示了所有已载入的项目的列表。选择其中的一个项目，将会在Packages视图中显示它所包含的包。选择一个包将会在Types视图中显示它的类型。选择一个类型可以在Members视图中显示其成员。在成员视图中选择一个方法或域成员将会在对应的编辑器中将该成员设置为高亮。

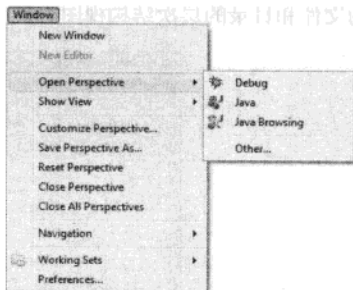


图1-4 打开Java Browsing透视图

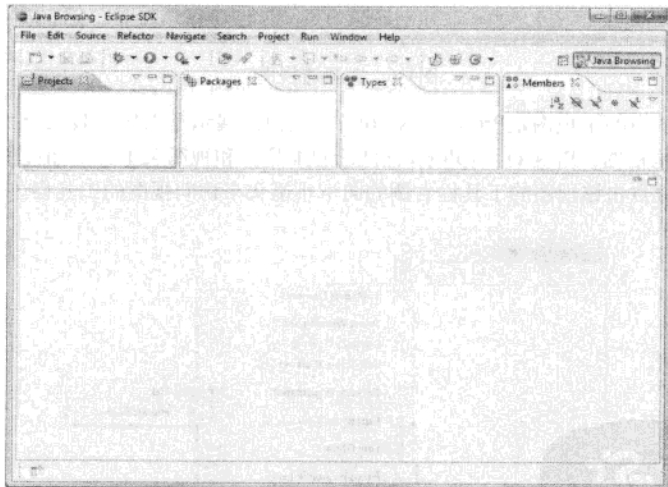


图1-5 Java Browsing透视图

提示 可以随意地拖曳单独的视图以设置成符合你自己的习惯。要获得编辑区域更多的竖直方向上的内容，可以考虑将四个视图竖直排列。另一个普遍的在一个透视图节省空间的办法是将项目（Projects）和包（Packages）视图合并成一个标签重叠区域，或将项目视图拖至预览视图栏。

2. Resource透视图

虽然Java透视图提供了Java开发工具，但是它不是很适合查看工作区的所有资源。你可以通过

Window > Open Perspective > Other... (图1-6) 命令打开Resource透视图。该透视图提供了你系统中的文件和目录的层次结构视图。

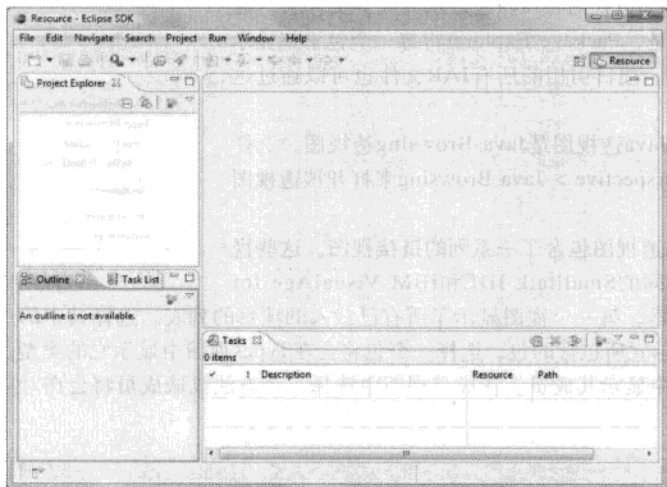


图1-6 资源透视图

Resource透视图中的主视图是Project Explorer。该视图展示了工作台载入资源（项目、文件夹和文件）的层次结构视图。Project Explorer有自己的工具栏和视图菜单，从而提供了不同的查看和过滤选项。可以通过点击该视图的工具栏右侧的向下小箭头来访问视图的工具栏。

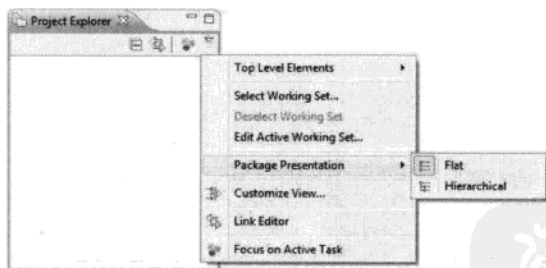


图1-7 项目资源管理器视图

3. Debug透视图

到目前为止，所介绍的透视图都是为编写代码或编辑资源而优化的。然后你会最经常用到的透视图是Debug透视图。你可以通过Window > Open Perspective > Debug命令打开它（图1-8）。

正如同它名字所表示的，Debug透视图用于调试项目。它可以较好地发现并更正Java代码中的运行时错误（runtime error）。你可以在代码中进行单独语句步进调试，设置断点，监视独立变量的值。在1.10.1节中有更详细的讨论。

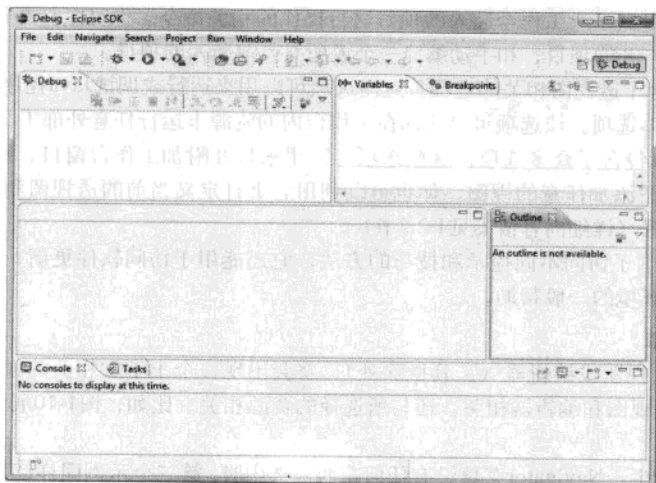


图1-8 Debug透视图

1.2.2 操作

除了组成显示区域的视图和编辑器之外，Eclipse还包含一些主菜单和上下文菜单和工具栏按钮。这些代表了系统中不同的可用命令或操作。

1. 主菜单

Eclipse的菜单栏包括了10个主菜单：File、Edit、Source、Refactor、Navigate、Search、Project、Run、Window和Help（图1-9）。其他菜单项根据你安装的插件工具和你所使用的透视图和视图而呈现。

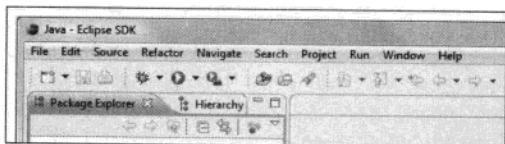


图1-9 Java透视图的菜单栏和工具栏

- File菜单提供创建新资源；保存、关闭和打印资源；刷新资源相关的磁盘文件；导入导出资源；监视资源属性和退出工作台等操作。
- Edit菜单提供用于在编辑区域打开的资源的相关操作，包含了标准功能，如剪切、复制、粘贴、删除、全选、查找和替换等。
- Source菜单提供操作当前源代码的命令，如添加和删除块声明、左移位和右移位、格式化、管理导入等。
- Refactor菜单提供了重构选定Java元素的命令。重构包括重命名域成员和方法、移动元素、提取方法和本地变量、将变量转换为域成员等。
- Navigate菜单提供了在工作台载入的资源间浏览的操作。它可以让你细分资源，在资源中导航，就如同一个Web浏览器一样。
- Search菜单提供了全工作台范围内搜索工具，如全局文件搜索、帮助搜索、Java搜索和插件搜索。我们将在后续章节深入讨论搜索。

- **Project**菜单提供了对工作台装入的项目进行操作的相关命令。你可以打开任意已关闭的项目，关闭任意已打开的项目，和手动编译一个或所有当前装入的项目。
- **Run**菜单包含了透视图相关的选项。这些选项可以用来运行或调试Java程序。它还包含了一个**External Tools**选项。该选项可以让你在工作台内的资源上运行任意外部工具。
- **Window**菜单包含了众多选项。这些选项可以用来打开附加工作台窗口，打开不同的透视图，给当前透视图添加任意的视图。你也可以利用它来自定义当前的透视图和访问当前整个工作台的参数设置（详细内容请参见1.2.2节）。
- **Help**菜单提供了访问不同提示和技巧的方法。它还能用于访问软件更新，当前工作台的配置信息和整个环境的一般帮助。

2. 上下文菜单

右键点击任意视图或编辑器（工具栏除外），将会出现一个上下文相关的弹出菜单。该菜单内容不仅与所点击的视图和编辑器相关，还与所选择的资源相关。比如，图1-10展示了3个示例上下文菜单。

第一个示例产生于Navigator菜单没有任何东西被选中时。第二个示例同样产生于Navigator菜单，但选中了一个Java文件。第三个示例展示了当在Package Explorer视图中的Java文件被选中时的上下文菜单。请注意一些选项，如Refactor子菜单，仅在某些视图，特定条件下才出现。

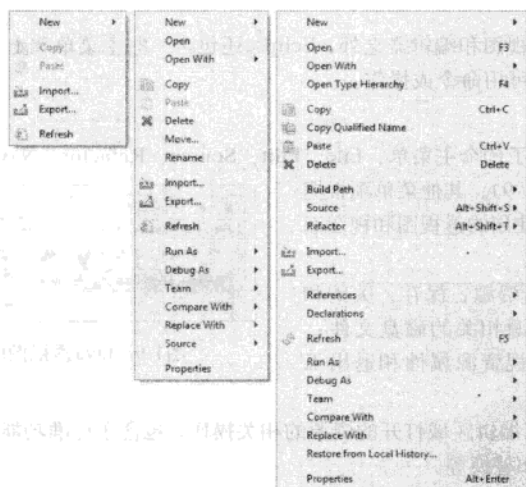


图1-10 上下文菜单

3. 工具栏

与右键点击视图出现的上下文菜单类似的是，工具栏项也是上下文相关的，它取决于当前所使用的透视图和所聚焦的编辑器。标准的、常用的选项出现于工具的前部，而编辑器相关的选项出现于后部。当使用Resource透视图时，可用的标准工具栏按钮包括创建新文件，保存和打印资源，运行外部工具，使用搜索功能和浏览最近打开的资源（图1-11）。

切换至Java透视图（图1-9）将出现几个新的图标。包括运行或调试Java程序，创建新的Java项目、包和文件。

4. 自定义可用操作

你可以对工具栏和主菜单栏的选项进行有限的控制。许多命令是命令组（command group）的一部分。命令组也称为操作集（action set）。操作集可以通过使用Customize Perspective选择是否启用。为了自定义当前透视图，可以选择Window > Customize Perspective...命令。这将会打开Customize Perspective对话框（图1-12）。工具栏和菜单命令组在Commands页面显示。选中所有你想要保留的命令组，其他的均不要选中。使用对话框的Shortcuts页来自定义New、Open Perspective和Show View菜单的条目。

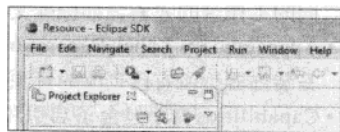


图1-11 Resource透视图的菜单栏和工具栏

1.3 设置Eclipse

上一节简要地介绍了自定义当前透视图。本节将涉及更多通过更改首选项来自定义Eclipse环境的细节。可以通过选择Window > Preferences...命令来更改Eclipse的首选项。该命令将打开Preferences对话框（图1-13）。数十个独立的首选项页面名称以树形结构显示在对话框的左侧。工作台的主要首选项在General组。Java首选项在Java组。在对话框的顶部，有一个十分方便的过滤文本框可以较快地找到特定的首选项页面。

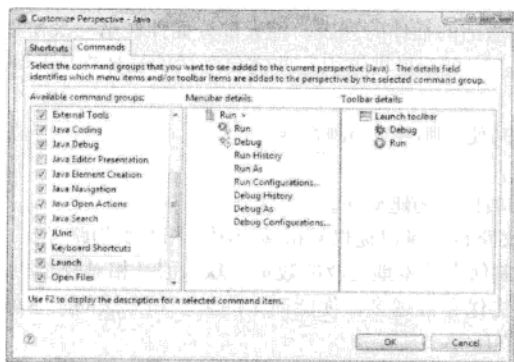


图1-12 自定义透视图对话框

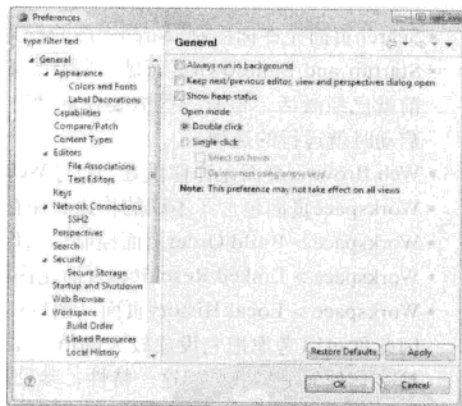


图1-13 首选项对话框

数百项不同的参数可以通过Preference对话框进行设置。更改一个参数值，再点击Apply按钮会应用更改并可以继续设置其他参数。点击OK按钮应用更改并关闭对话框。Restore Defaults按钮将使用系统默认值更改当前页面的首选项。

1.3.1 工作台首选项

Eclipse的大部分常用参数可以在General类别中找到。几个主要内容包括：

- General页面决定打开一个资源是单击还是双击，和堆状态指示器是否在工作台的状态栏显示。
- Appearance页面决定视图和编辑器的选项卡是出现在顶部还是在底部。

- Appearance > Colors and Fonts页面提供自定义颜色和字体的选项。这些选项可以用于许多不同的工作台的元素，如标准文本字体、对话框字体，头部字体，错误颜色等。
- Appearance > Label Decorations页面提供了增强某项内容的图标和标签的选项。比如，CVS标签装饰器给所有已更改的资源添加“>”前缀。
- Capabilities页面包含了启用和禁用不同的功能的选项。这些功能允许你作为组启用和禁用不同的产品功能。
- Compare/Patch页面允许你控制文本比较视图的行为。
- Content Types页面包含了将不同内容种类和不同的文件类型相关联的选项。
- Editors页面包含了一系列用来控制编辑器打开关闭方式和一次可以打开的编辑器数量的选项。
- Editors > File Associations页面将不同的编辑器类型（包括内部编辑器和外部编辑器）和不同的文件类型关联起来。比如，如果你想将Adobe Dreamweaver和HTML文件关联起来，你可以在这里进行设置。
- Editors > Text Editors页面包含了控制编辑器外观的选项，如可视内容的行数、当前行高亮、不同项的颜色和注释。
- Keys页面提供了自定义与系统中命令所绑定的键的选项。它包含了一个预定义的标准键绑定集和一个Emacs键绑定集。
- Network Connections页面允许你设置可以由多个插件重用的Internet连接的代理。
- Perspectives页面允许你选择默认透视图和新的透视图是在当前窗口还是新窗口中打开。
- Search页面允许你控制Search视图的行为。
- Startup and Shutdown页面显示所有需要早期激活的插件的列表。绝大部分插件在第一次使用前就已经激活，但有一些需要在启动时激活。本页面提供组织这些需要激活的插件在Eclipse启动时就运行的选项。
- Web Browser页面允许你设置当打开Web页面时使用哪种Web浏览器。
- Workspace页面包含了不同的构建和保存选项。
- Workspace > Build Order页面控制你工作区的项目的构建顺序。
- Workspace > Linked Resources页面允许你定义路径变量以提供对链接资源的相对引用路径。
- Workspace > Local History页面（图1-51）控制保存的本地更改的数量。默认值是相对较小的，因此你可以考虑把它设成较大的值。本地历史保存得越多，你能回滚的类型和方法的版本就越多（如何更好地使用这一特性，参见1.7.4节）。

1.3.2 Java首选项

Eclipse所包含的Java开发工具的参数设置可以在首选项的Java类别找到。一些主要的参数项包括：

- Java页面提供控制不同Java视图和编辑器的行为的选项。
- Appearance页面控制在不同Java视图中的Java元素的外观。
- Build Path > Classpath Variables页面（图1-14）提供定义新的classpath变量的选项。这些classpath可以被添加至某个项目的classpath。
- Code Style > Formatter页面（图1-39）控制用于格式化Java代码的Eclipse Java代码格式化器的相关选项，包含有控制大括号的位置、新行、行的长度和空格使用的选项。

- Code Style > Code Templates页面定义了命名约定和生成代码包含的类型、方法、域、变量和参数的默认注释。
- Compiler页面提供了控制不同编译和构建路径问题的严格程度的选项，也包含了不同JDK依赖性的选项。
- Editor页面控制Java编辑器元素外观的选项（如括号匹配、打印边框和当前行高亮）、Java语法的高亮颜色（图1-38）、代码辅助的行为和外观，以及问题注释。
- Editor > Templates页面提供了定义和编辑不同的Javadoc和Java代码模板（模板是指在用户编写的代码中频繁出现的一般源代码样式）。
- Installed JREs页面提供了指定工作台所使用的JRE的选项。

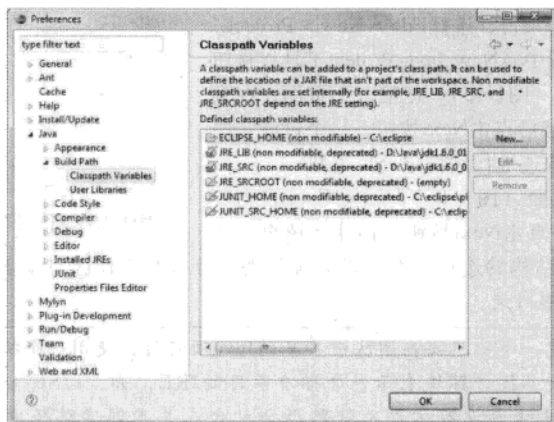


图1-14 Classpath变量首选项页

1.3.3 导入与导出首选项

设置多个Eclipse工作区或升级Eclipse版本相对是比较困难的。难度主要在于将一个版本的工作区参数迁移至另一个版本。同样地，设置多个用户工作区的共同设置，如代码格式化首选项和classpath变量设置，同样有可能相当困难。

Eclipse Export与Import向导包含了Preferences选项以帮助解决这一问题。选择File > Export..., 然后选择Preferences将打开一个向导。该向导用于提示输入参数导入文件（一个.epf文件）的名字，并记录输入的文件中的所有非默认的参数设置。选择File > Import..., 然后选择Preferences将打开另一个向导。该向导用于导入一个参数文件。该向导提供不同级别参数导出粒度的选项。你可以导出所有工作区的参数，或导出特定的参数。

这种导入导出参数的机制还称不上是理想的。这是因为在处理诸如classpath变量（导出时将使用硬编码路径而不是工作区相关的路径）等不同类型的参数与代码模板（完全不导出）上还存在一定不足。

1.4 创建项目

本章之前的内容介绍了Eclipse工作台并展示了一些自定义环境的方法。然后，我们就将真正开

始使用Eclipse来做些事情了。这一节将带领你一步步地创建第一个Eclipse项目。

在基本的Eclipse环境中，可以创建三种不同类型的项目，简单项目、Java项目和插件开发项目。

1) 简单 (Simple) 项目，正如名字所表达的，是最简单的Eclipse项目。它们可以包含任意类型的资源，包括文本文件、HTML文件等。

2) Java项目，用于保存Java源代码和Java应用程序相关资源。下一节将描述如何创建Java项目。

3) 插件开发 (Plug-in development) 项目，用于创建Eclipse插件。这将是本书主要关注的项目类型。第2章详细说明了如何创建一个插件项目。

1.4.1 使用新建Java项目向导

要创建一个新的Java项目，选择File > New > Project...命令，或点击Java视图中New Java Project工具栏按钮来打开New Project向导 (图1-15)。在第一个页面，在列表中选择Java Project，然后点击Next按钮。一个过滤字段出现在向导顶部，可以快速找到特定项目类型。

在向导的第二个页面 (图1-16)，输入项目的名字 (如“First Project”) 并点击Next按钮。请注意该页面还包括了指定项目位置和结构的选项。默认地，项目将会存放在工作区目录下并使用src和bin文件夹作为源文件和类文件的根目录。

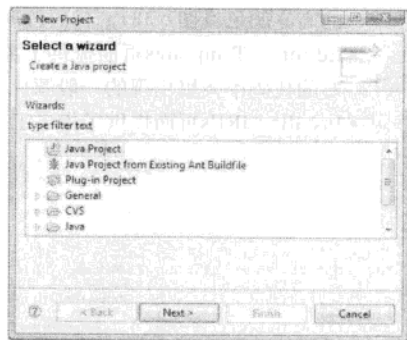


图1-15 新项目向导——选择项目类型

提示 当你准备创建一个你想要在团队中共享的Java项目时，使用一个执行环境代替一个特定的JRE是较好的办法。执行环境表示包含标准条目的JRE，如“J2SE-1.4,” “J2SE-1.5,” 和“JavaSE-1.6.” 这意味着没有文件系统路径会包含在共享创建路径中。JRE可以在Java > Installed JREs > Execution Environments首选项页分配给执行环境。

向导的下一页面 (图1-17) 包含了为Java项目进行构建路径设置的内容。在Source选项卡可以添加

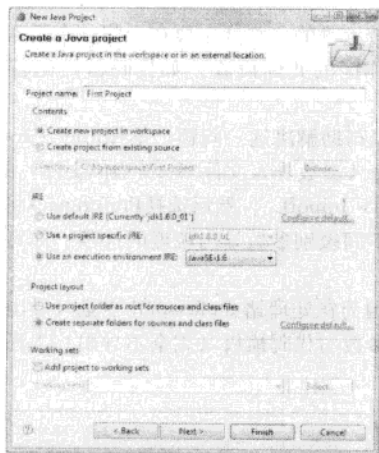


图1-16 新项目向导——给项目命名

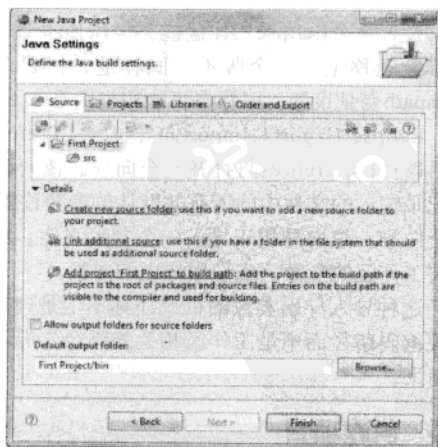


图1-17 新建项目向导——指定Java构建设置

源代码文件夹，以作为包含Java文件的包的根目录。传统上，源代码文件夹一般单独放置在一个名为src的文件夹中，而编译器输出文件则放置在名为bin的目录。将这些类型的文件分文件夹单独放置将会使得构建过程更简单。使用Java > Build Path首选项页来创建新Java项目时所使用的默认目录名称。

Projects选项卡允许你通过选择工作区的其他项目来设置本项目的依赖项。Libraries选项卡可以添加JAR文件（既可在工作区内也可在工作区外的文件系统中）。最后一个选项卡Order and Export，控制构建路径元素的顺序，以及它们对于依赖本项目的其他项目是否被导出和可见。

在页面的底部的Default output folder字段，用来指定存放编译输出的默认位置。当点击Finish按钮，新的项目将被创建，并出现在Package Explorer视图或Navigator视图中，这取决于哪一个透视图是活动的。

Eclipse不同版本的区别 正如在后三个截图中所看到的那样，Eclipse 3.4和较老版本存在一些小区别。在绝大多数时候，区别不会影响到你使用本书。然而，当区别是相关的或令人感兴趣的时候，将会高亮该区别。

1.4.2 .classpath和.project文件

除了创建项目本身，还创建了两个附加文件——.classpath文件和.project文件。默认地，这两个文件如同其他以“.”开头的文件一样，通过过滤器查看的时候是隐藏的。

为了显示这两个文件，在Package Explorer的下拉视图菜单中选择Filters...命令（图1-18），在Java Element Filters对话框中（图1-19），取消选中.* resources过滤器，并点击OK按钮。

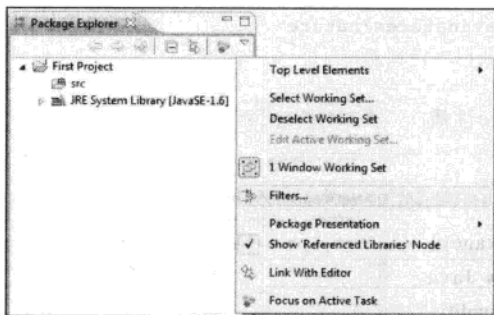


图1-18 过滤器菜单

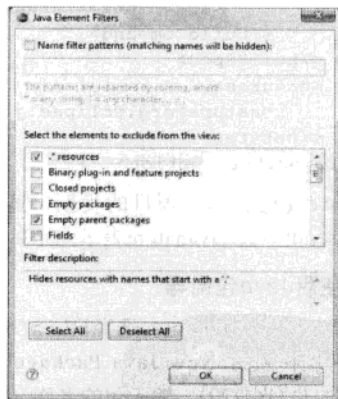


图1-19 过滤器对话框

.classpath文件存储了项目的Java构建路径。在你刚创建的项目中，它看起来和以下内容类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
  <classpathentry kind="src" path="" />
  <classpathentry kind="con"
    path="org.eclipse.jdt.launching.JRE_CONTAINER/
    org.eclipse.jdt.internal.debug.ui.launcher.StandardVMType/
    JavaSE-1.6" />
```

```
<classpathentry kind="output" path="" />
</classpath>
```

相对于直接编辑.classpath文件，Eclipse提供了一个更友好的方法。右键点击项目，选择Properties。在Properties对话框中，选择Java Build Path将显示一个类似于图1-17的界面以编辑项目的classpath。


Java Build Path “Java Classpath”是一个用于描述编译时和运行时所使用的classpath的通用短语。在Eclipse中，编译时classpath被称为Java构建路径。当运行或调试Java程序代码时，运行时classpath由启动配置（参见1.9.2节）决定。当在开发Eclipse插件时，运行时classpath由插件manifest文件中的依赖性声明决定（参见2.3.1节）。

.project文件提供了项目的完整描述，可以用来在工作区中它被导出后再次导入时重新创建项目。你的新项目应该和以下类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
  <name>First Project</name>
  <comment></comment>
  <projects>
  </projects>
  <buildSpec>
    <buildCommand>
      <name>org.eclipse.jdt.core.javabuilder</name>
      <arguments></arguments>
    </buildCommand>
  </buildSpec>
  <natures>
    <nature>org.eclipse.jdt.core.javanature</nature>
  </natures>
</projectDescription>
```


nature标记表示该项目的类型。这里的nature性质org.eclipse.jdt.core.javanature表示它是Java项目。

1.4.3 使用Java包向导

为了创建一个Java包，选择File > New > Package或点击工具栏按钮  New Java Package以打开New Java Package向导（图1-20）。输入包的名称（如“com.qualityeclipse.sample”），然后点击Finish按钮。

请注意新建的包的名称的左侧的图标是空的，表示该包是空的（图1-21）。当一个或多个Java文件添加至该包中时，图标会着上颜色。

1.4.4 使用Java类向导

为了创建一个Java类文件，选择File > New > Class或点击工具栏的  按钮以打开New Java Class向导，如

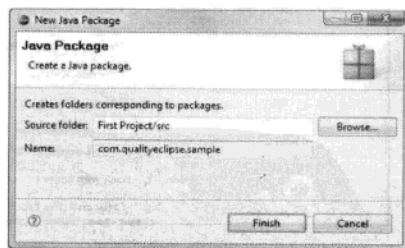


图1-20 新建Java包向导

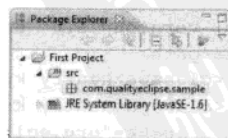


图1-21 包管理器中的新Java包

图1-22所示。输入类的名字（如“HelloWorld”），选中public static void main(String[] args)单选框，然后点击Finish按钮。请注意向导展示了创建一个新类的许多附加选项，包括它的超类、接口和默认初始方法。



图1-22 新建Java类向导

该过程创建了一个新的Java类（图1-23）。HelloWorld.java条目文件，代表文件自身。扩展该项将显示代表类及其单独的“main”方法的元素。请注意包名字旁边的图标是着色的，表示它不再是空的了。

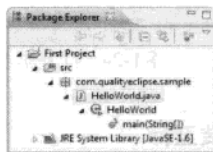



图1-23 包管理器中的新Java类

1.5 导航

Eclipse包含了一些用于在系统内快速导航和查找信息的工具。本节讨论一些可以从Eclipse Navigate菜单访问的工具。

提示 许多第三方插件为Eclipse提供了不同的导航的增强功能（见附录A）。比如，CodePro提供了一个Java History视图，该视图跟踪你所访问的所有Java文件。它还提供了一个Modified Type视图，该视图跟踪所有你更改过的类型。

1.5.1 打开类型对话框

Open Type对话框用于在系统内的任意Java类文件间进行快速跳转。可以通过Navigate > Open Type...命令（Ctrl+Shift+T），或点击工具栏的  Open Type按钮打开该对话框。然后输入你想要寻找的类型的名称。名称输入框可以使用通配符，将会显示一个所有匹配输入样式的类型的列表。对话框对驼峰样式（CamelCase）也提供了支持，因此输入“NPE”将会找到类NullPointerException。如果文本框没有输入任何内容，对话框将显示过去找到的类型的列表（第一次使用该对话框时，对话框将不显示任何内容）。

从列表中选择你想要的类型，然后点击OK按钮。这样就可以在编辑器中打开该类型。如果有多个的类型匹配你输入的名字，包名称限定符将会在类型名称右侧显示。

1.5.2 类型层次结构视图

Type Hierarchy视图显示给定类型的超类和子类（图1-25）。该视图也包含显示类型的超类型层次结构（包含超类和实现的接口）还是子类型（包含子类和接口实现者）层次结构的相关选项。

Type Hierarchy视图可以由几种方法访问。最简单的方法是在编辑器中选择类型的名称，然后选择Navigate > Open Type Hierarchy命令（也可以使用F4键）。另外，还可使用Navigate > Open Type in Hierarchy...命令（Ctrl+Shift+H）来打开Open Type对话框，如图1-24所示。

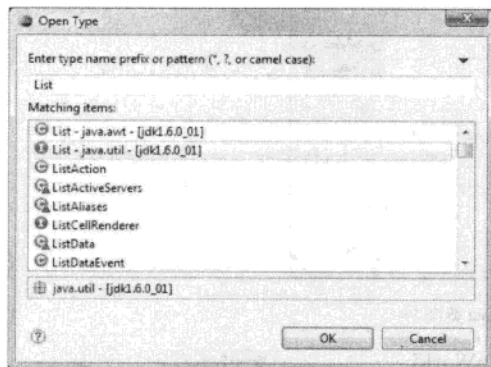


图1-24 打开类型对话框

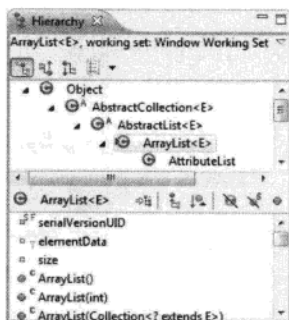


图1-25 类型层次结构视图

1.5.3 转至行

在文件内转到特定行，可以使用Navigate > Go to Line...命令（Ctrl+L）。这将打开一个提示用户输入的对话框。在该对话框中用户输入想要跳转至的行号（图1-26）。点击OK可以跳转至编辑器中相应的行。

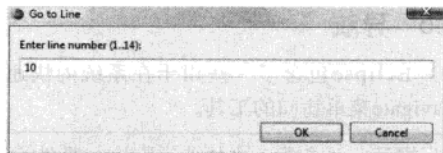


图1-26 行号提示输入框

1.5.4 大纲视图

Outline视图显示了选定编辑器的结构元素的大纲。该视图的内容取决于当前所使用的编辑器的类型。比如，当编辑一个Java类时，大纲视图该Java类的类、字段和方法（图1-27）。

Java Outline视图包含了一些控制在大纲中显示元素的选项。对于隐藏字段、静态成员、非公有成员和本地类型，均具有过滤器。此外，该视图还具有排列成员的选项（默认是以定义顺序显示），和追溯至最高级类型的选项（一般地，大纲首先以文件级别开始）。

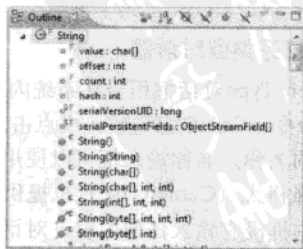



图1-27 大纲视图

1.5.5 快速访问

为了快速访问视图、命令、菜单、首选项页面和向导，可以使用Quick Access对话框。该对话框可以通过Window > Navigation > Quick Access访问 (Ctrl+3)。在过滤器字段中开始输入以查看匹配结果，使用方向箭头来选择匹配项，然后按回车键执行命令或打开相应的视图、透视图或向导。

1.6 搜索

除了在Navigate菜单中可使用的导航工具之外，Eclipse包含了一些功能强大的搜索工具。这些工具可以在Search菜单找到。通过Search > Search...命令 (Ctrl+H)，可以打开Eclipse Search对话框。也可以通过点击  Search工具栏按钮来打开该对话框。该对话框包含一些不同的搜索工具，包括了File Search、Task Search、Java Search和Plug-in Search。这其中两个最重要的工具是File Search和Java Search。

1.6.1 文件搜索

Search对话框的File Search选项卡 (图1-28) 提供了在工作台内查找任意文件的方法。可以通过按文件名和按它们所包含的文本进行搜索。为了搜索包含特定语句的文件，将该语句输入Containing text字段。该字段支持不同的通配符，如“*”用于匹配任意字符串，“?”用于匹配任意字符。默认地，该搜索是对大小写敏感的。为了将其设置成大小写不敏感的，取消选中Case sensitive选项。要使用正则表达式进行复杂文本搜索，请选中Regular expression选项。

为了通过文件名搜索文件，可以不填Containing text字段。为了限制搜索特定类型的文件或包含特定命名规则的文件，在File name patterns字段输入相关内容。

Scope字段提供了另一种更严格的搜索方法。Workspace范围包含了整个工作区，Working set范围将搜索限制在包含于指定工作集内的文件。Selected resources范围将搜索限制在仅搜索在活动视图内的文件 (比如，Navigator视图或Package Explorer视图)，而Enclosing projects范围将搜索限制在包含指定文件的项目。

比如，为了搜索所有包含文本“xml”的文件，在Containing text字段中输入该文本，并且不改变File name patterns字段和Scope字段。当准备好时，点击Search按钮以查找匹配的文件并在Search视图中显示它们 (图1-29)。点击Replace按钮而不是Search按钮将执行同样的搜索，但它将打开Replace对话框以让你输入替换文本。

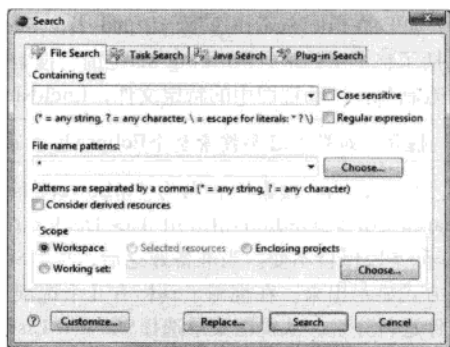


图1-28 文件搜索选项卡

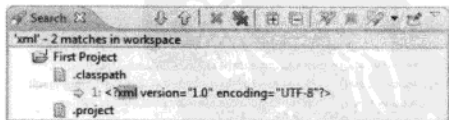


图1-29 文件搜索结果

提示 如果你打算搜索不同的Java元素，如类型、方法、字段等。Java Search选项比File Search选项功能更强大。

1.6.2 Java搜索

Java Search选项卡（图1-30）用于查找Java元素，如类型、方法、构造函数、字段和包。你可以使用它来查找特定Java元素的声明、对元素的引用或元素的实现者（在Java接口中）。

为了搜索特定名称的元素，在Search string字段中输入该名称（支持通配符）。根据你所感兴趣的Java元素的类型，选择Type、Method、Package、Constructor或Field单选按钮。你可以更进一步地将搜索结果限制为Declarations、References、Implementors（Java接口）、Match locations（用来调整声明、表达式或搜索的参数类型）、All Occurrences、Read Access（用于字段）或Write Access（用于字段）。Search In字段允许你将搜索限定在你的项目资源、所依赖的资源、JRE和应用程序库。

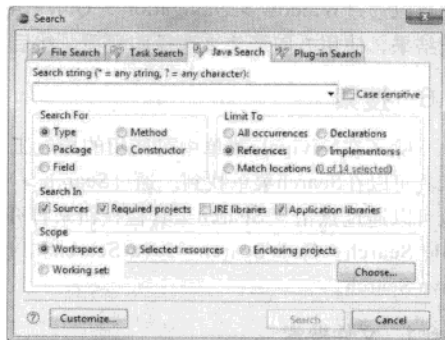


图1-30 Java搜索选项卡

如同在File Search选项卡中的一样，Scope字段提供了另一种限制搜索的方法。Workspace范围包括了整个工作区，Working set范围将搜索限制于一个特定的工作集，Selected resources范围将搜索限制与在活动视图中的特定文件，Enclosing projects范围将搜索限制于包含指定文件的项目。

提示 如果你想要搜索整个Eclipse插件源代码（参见21.1节），可以考虑创建一个引用项目。

比如，为了搜索所有名为“toLowerCase”的方法，在Search string字段中输入该字符串，选中Search For > Method和Limit To> Declarations单选按钮。选中Search In > JRE Libraries单选框，Scope字段保持不变。当准备好之后，点击Search按钮以查到匹配该名称的方法，并在搜索视图以层次方式显示出来。在视图工具栏有几个选项可以用于分组搜索结果，可以通过项目、包、文件或类文件进行分组。从视图菜单选择Show as List命令来单独列表查看搜索结果（图1-31）。

在任意搜索结果上双击将打开一个包含该搜索结果的文件的编辑器，并将文本中的匹配结构设置为高亮，并在编辑器的左侧空白处放置一个搜索标记（也称为标记栏，或左侧竖直标尺）（图1-32）。点击Search视图中的Show Next Match或Show Previous Match按钮（上下箭头）将会选择下一个或上一个匹配项（如果需要则将会在新编辑器中打开不同文件）。你可以通过使用Search视图中的上下文菜单继续搜索（“drill-down”）。

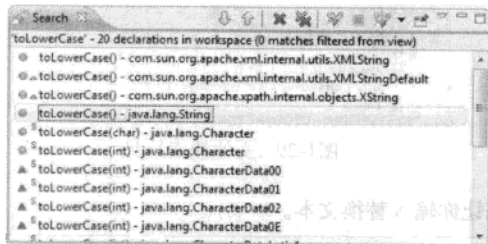


图1-31 Java搜索结果

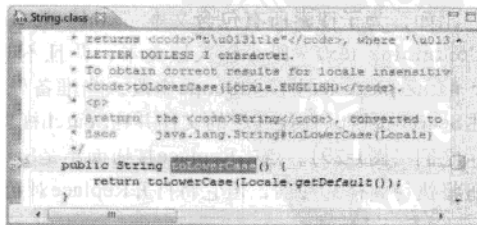


图1-32 显示搜索匹配和搜索标记的编辑器

Java插件搜索

当查找Java类或方法时，搜索范围包括了工作区和工作区内的项目引用的所有库和插件。这样会导致在项目没有引用的插件中查找类变得困难。为了解决这一问题，你可以在搜索范围明确添加那些没有被任何项目所引用的插件。打开插件视图（图2-25），在该视图中选择一个或多个插件，然后右键点击并选择Add to Java Search。这样就把所选择的插件的类和方法添加至所有随后的Java搜索的范围。

提示 选择Add to Java Search将创建一个新的项目。该项目将被命名为“External Plug-in Libraries”。该项目将被作为插件添加至原项目。为了使该项目在Package Explorer视图中可见（参见1.2.1节），在Package Explorer中下拉视图菜单，选择Filters... Uncheck External plug-ins library project，然后单击OK按钮。

此外，你也可以通过创建引用项目向搜索范围添加插件。引用项目本身是一个插件项目，它包含了对一个或多个在搜索范围内的插件的引用。想了解更多有关引用项目的信息，请参见21.1节。

1.6.3 其他搜索菜单选项

Search菜单包含了一些专门的、易于使用的Java搜索命令。这些命令复制了搜索对话框中的Java Search页的选项（图1-33）。

无论是在视图中还是在Java编辑器中选择一个Java元素，然后选择Search > Declarations命令都会会查找到工作区内所有具有匹配声明的元素，包括当前项目内的、当前类型层次结构内的或指定工作集内的。类似地，选择Search > References命令将会查找到所有使用该元素的地方。Search > Implementors、Search > Declarations和Search > Write Access命令也具有类似的工作方式。请注意，类似的命令也可以在Java编辑器和Search视图中的上下文菜单找到。

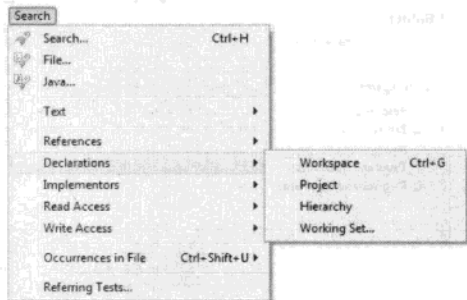


图1-33 专门的Java搜索命令

1.6.4 工作集

在之前的内容中，我们已经数次提到了工作集。它们用来创建一个元素组，以作为不同视图（如Navigator和Package Explorer）中的过滤器，或在Search对话框以及任意搜索菜单中的搜索范围。当你拥有一个很大的、包含许多项目的工作区时，你会发现工作集特别有用。这是因为它们限制了代码的范围，并且让许多任务变得更简单。

为了选择工作集或创建一个新的工作集，选择File Search对话框中的Scope > Working Set，然后点击Choose按钮。这样将打开Select Working Set对话框（图1-34）。想要使用一个已有的工作集，从列表

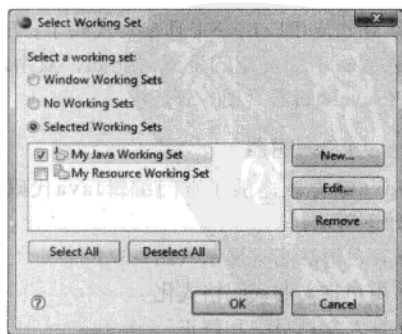


图1-34 选择工作集对话框

可以单击Edit按钮。

提示 Eclipse具有选择多个工作集的功能。这样会产生一个新的虚拟工作集，它包含了所有已选定工作集的结果。这样将使得创建多个、良好粒度的工作集变得更容易。然后，它将把这些选定的工作集以不同方式联合起来。

点击New...按钮以创建一个新的工作集。这样将打开New Working Set对话框（图1-35）。你可以创建五种不同类型的工作集：资源、断点、Java、任务和插件工作集。选择你想要创建的工作集类型，然后单击Next按钮。

New Working Set对话框的下一页面使定义新的工作集变得更容易（图1-36）。在Working set name字段输入名称，然后在Workspace content列表中选择内容，将它添加至Working set content列表。使用Add和Remove按钮可以移动元素。点击Finish按钮将关闭New Working Set对话框，并将新的工作集添加至Select Working Set对话框。

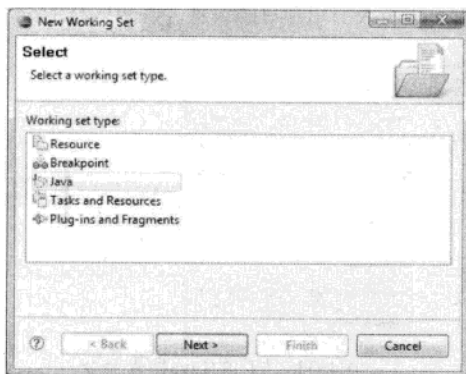


图1-35 新建工作集对话框

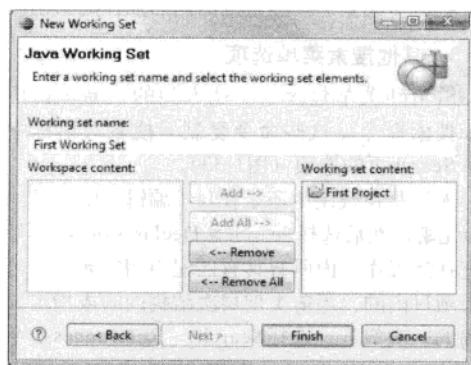


图1-36 定义新工作集

1.7 编写代码

当你的第一个Java项目已经创建好并且你已经尝试了几种在系统内导航与查找项目的方法之后，是时候开始使用Eclipse工具来写代码了。Eclipse使用一些不同编辑器，包括内部编辑器和外部编辑器，来编辑不同类型的资源。比如，双击Java文件，将打开Java编辑器（图1-37）。

1.7.1 Java编辑器

Java编辑器提供了专门编辑Java代码的许多特性，如下所示：

- 彩色语法高亮显示（图1-37）
- 用户定义的代码格式化
- 导入文档组织和修正
- 上下文相关的代码辅助

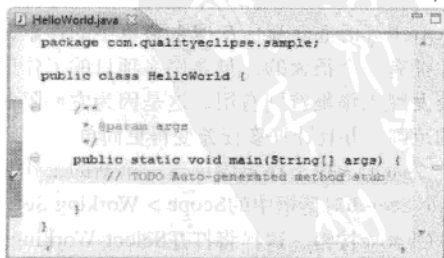



图1-37 Java编辑器

- “快速修复”问题自动更正

提示 许多以前的VisualAge for Java的用户喜欢该IDE的每次仅显示一个方法而不是整个Java文件的功能。Eclipse的Java编辑器同样支持该功能，可以通过工具栏的  Show Source of Selected Element Only按钮开启。为了让该功能发挥作用，你必须将焦点放置于一个编辑器上。这是由于该按钮直到你开始编辑代码时才能点击。这是Eclipse中的应作为一个工作区参数而不是工具栏按钮的选项之一。

1. 彩色语法高亮显示

彩色语法高亮显示功能控制Java代码以何种方式呈现。独立的颜色和字体样式（普通或粗体）控制可以用于多行或一行注释、关键字、字符串、字符、任务标记和Javadoc元素。这一选项在Java > Editor > Syntax Coloring首选项页进行设置（图1-38）。

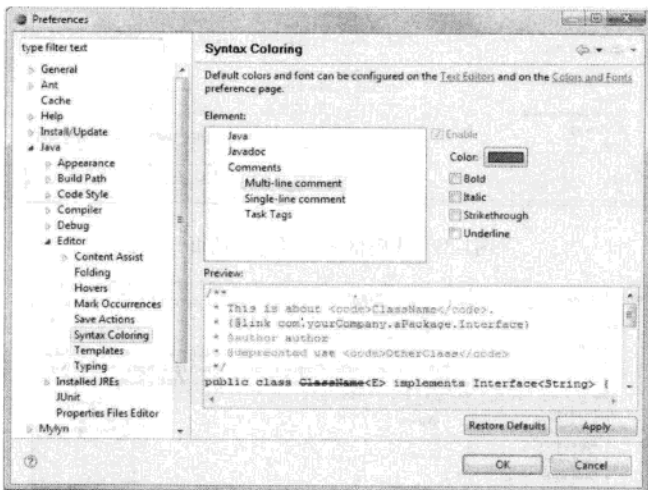


图1-38 语法色彩首选项页

2. 用户定义的代码格式化

代码格式化功能控制点击Source > Format命令时Java代码的格式化方式。它提供了一些用于控制大括号位置、新行、行的长度和空白使用的选项，这些可以在Java > Code Style > Formatter首选项页进行设置（图1-39）。

提示 还有其他代码格式化工具可以使用，包括Eclipse的Jalopy集成（jalopy.sourceforge.net）。

3. 组织Java导入语句

导入组织功能提供一种简易清理Java文件中的导入语句的方法。可以使用Source > Add Import添加新的导入，已有的导入可以使用Source > Organize Imports进行清理。Java > Code Style > Organize Imports首选项页（图1-40）提供默认导入语句顺序和使用通配符导入过程中的阈值的相关设置。

提示 将阈值设为1将立即导入所有“.”的包。如果保留默认值99，将根据编码习惯来分别导入不同类型。

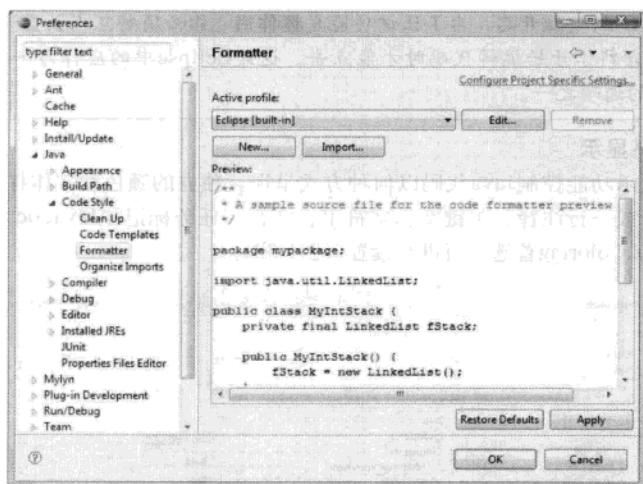


图1-39 代码格式化器首选项页

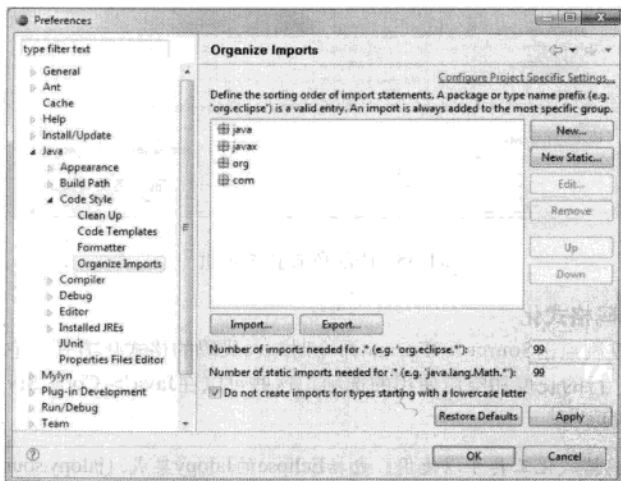


图1-40 组织导入首选项页

4. 上下文相关的内容辅助

上下文相关的内容辅助功能可以帮助大幅提高Java代码编写速度。它可以完成类名称、方法名称、参数名称等。驼峰样式，如NPE，将会被扩展为类的全名，比如NullPointerException。

为了使用该功能，将鼠标放置于你代码需要建议的位置，然后可以通过选择Edit > Content Assist > Default命令或按住Ctrl键后按下空格键。这将弹出代码辅助窗口（图1-41）。

提示 如果代码辅助窗口无法打开，取而代之的是发出系统声音，请检查Java创建路径，然后检查代码。这是因为代码包含了很多错误以致编译器无法明白代码的含义。请记住，Java编译器一直在后台工作！

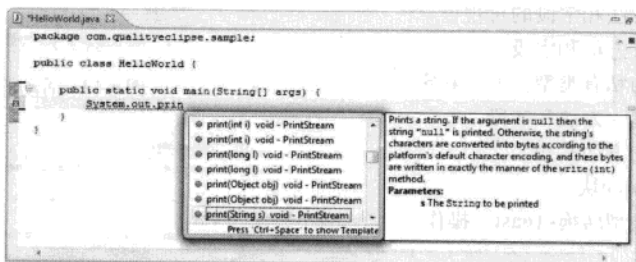


图1-41 活动的内容辅助

Java > Editor > Content Assist首选项页（图1-42）提供了一些内容辅助功能被激活时如何工作的选项。

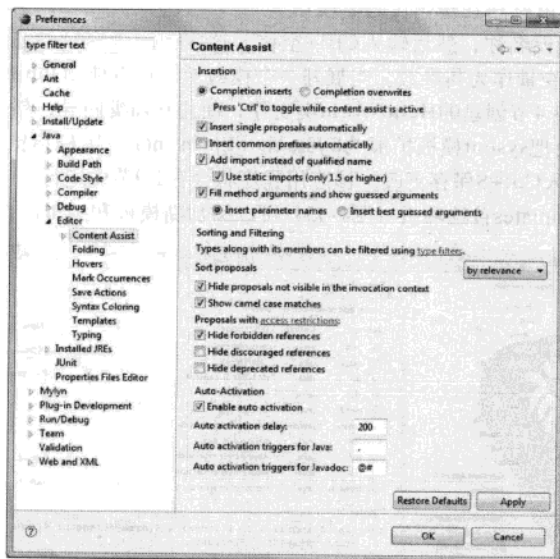


图1-42 内容辅助首选项页

5. “快速修复”自动问题更正

“快速修复”功能提供了一种简易修复Java编辑器中存在问题的方法。任何时候发现的问题都能

被修复。一个电灯图标在标记栏（左侧竖直标尺）显示。点击该图标或选择Edit > Quick Fix命令将弹出一个快速修复窗口（图1-43）。从列表中选择合适的选项将修复Java源代码。

有数十个不同的快速修复可以使用，包括：

- 更正丢失的或不正确的包声明
- 移除未使用的或重复的导入
- 改变类型、方法和字段的可视性
- 重命名类型、方法和字段
- 移除未使用的私有类型、方法和字段
- 创建新的类型、方法和字段
- 修复不正确的方法参数
- 添加或移除catch块
- 添加必需的类型转换（cast）操作

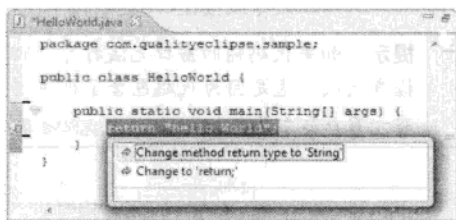


图1-43 活动的快速修复

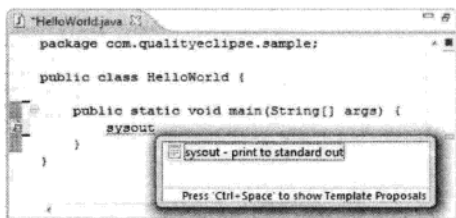


图1-44 活动的模板弹出窗口

1.7.2 模板

模板是在用户编写的代码中经常出现的通用源代码样式。Eclipse拥有数十种内建模板，添加新模板也比较容易。

为了使用模板，将鼠标指在Java代码要使用模板的地方，开始键入模板名称，然后输入Ctrl+空格。这将弹出内容辅助窗口（图1-44）。请注意，一些模板包含用户定义变量作为其参数。当展开一个模板后，可以使用Tab键在参数间移动。

举个例子，打开1.4.4节创建的HelloWorld类文件，使用Java类向导，然后输入“sysout”，然后输入Ctrl+空格。这将会把sysout模板扩展为System.out.println();，鼠标指针放置于圆括号中。输入“Hello World”，然后输入Ctrl+S保存更改。该应用程序将会在1.9节中运行。

Java > Editor > Templates首选项页（图1-45）可以添加新模板和编辑已有的模板。

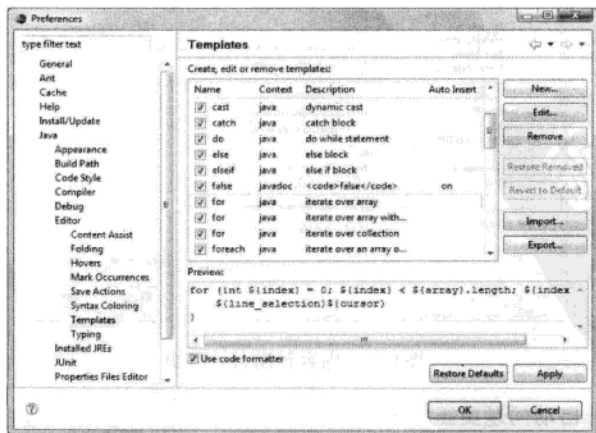


图1-45 模板首选项页

为了添加新的模板，点击New按钮以打开Edit Template对话框（图1-46）。在Name字段中输入样式的名称，在Description字段中输入描述，在Pattern字段（请注意代码辅助是大小写不敏感的）输入样式。

Eclipse支持两种类型的样式，Java样式和Javadoc样式。从Context下拉列表中选择样式类型。Insert Variable按钮将弹出一个可以插入到模板中的变量的列表。点击OK按钮将模板添加至模板列表。

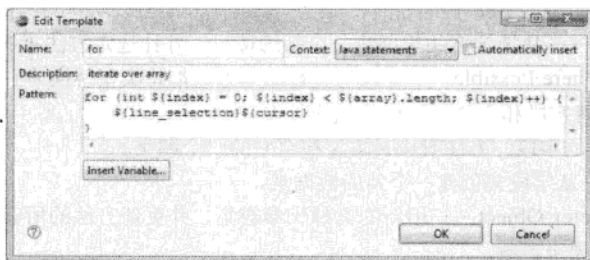


图1-46 编辑模板对话框

提示 一些第三方插件提供了增强模板（即Eclipse中样式）（见附录A）。

1.7.3 重构

重构（Refactoring）是在不改变项目的外部行为的前提下，改变一个软件系统以改进其内部结构和可重用性的过程。它可以清理代码以减少bug出现的几率。本质上，当开发人员进行重构，他们就是在改进代码的设计。Eclipse提供一个功能非常强大、相当复杂的重构工具集，可以让重构变得迅速、简单和可靠。

Eclipse重构命令可以通过两种途径访问。一种是从Java编辑器的上下文菜单，另一种是从主菜单的Refactor菜单（在Java编辑器打开时可用）。Refactor菜单（图1-47）包括超过二十个不同的重构命令，这些命令用于设置Java元素的一些方面，并更新工作区内所有对它的引用。

Eclipse支持很多的重构命令，包括：

- Rename——重命名Java元素。
- Move——移动Java元素。
- Change Method Signature——改变方法的参数（名称、类型和顺序）。
- Extract Method——创建一个基于当前方法中所选定文本的新方法，并用一个对新变量的引用替代选定文本。
- Extract Local Variable——创建新的本地变量，分配给选定的表达式，并用一个对新变量的引用替代选定文本。
- Extract Constant——从选定表达式创建一个static final字段。

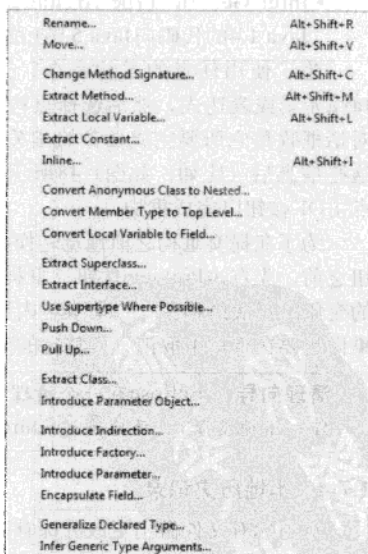


图1-47 重构菜单

- Inline——内联方法、常量和本地变量。
- Convert Anonymous Class to Nested——将匿名内部类转换为命名嵌套类。
- Convert Member Type to Top Level——将嵌套类型转换为最高级类型。
- Convert Local Variable to Field——将本地变量转换为字段。
- Extract Superclass——从一个具有共同超类的类集创建共同超类。完成该项重构后，所选定的类将成为创建的超类的直接子类。
- Extract Interface——从选定方法集创建一个新的接口，并让选定方法集实现该接口。
- Use Supertype Where Possible——在任何可行的地方，都将类型替换为它的一个超类。
- Push Down——将字段和方法从类移至它的子类。
- Pull Up——将类的字段、方法或成员类型移至它的一个超类。
- Extract Class——从字段集创建一个新的数据类。
- Introduce Parameter Object——用新的类替代参数集，并更新方法的所有调用者以将新类的一个实例作为值传递至参数。
- Introduce Indirection——创建代表选定方法的静态间接方法。
- Introduce Factory——使用对新工厂方法的调用来替代构造方法调用。
- Introduce Parameter——用参数引用替代表达式。
- Encapsulate Field——将所有对字段的引用替换为对该字段的getter和setter方法的引用，并在必要时创建这些方法。
- Generalize Declared Type——普通化变量声明、参数、字段和方法返回类型的类型。
- Infer Generic Type Arguments——为一个类、包或项目中的泛型引用推测类型参数。这在由Java 1.4的代码向Java 5.0迁移时尤为有用。

为了使用任意的重构命令，选择你想要重构的Java元素或表达式，然后选择重构命令。每一个重构对话框收集它所完成任务的相关信息。当你提供了这些信息后（比如，如图1-48所示的新方法的名称）。点击OK按钮以完成重构。

为了在提交重构之前预览转换内容，在点击OK按钮之前，先点击Preview按钮。重构预览视图把将发生的变化以层次结构在一个文本区内显示出来，并显示所改变代码的之前和之后的视图（图1-49）。如果你想要在重构中取消一个特定的变换，在树形列表中取消选中它。

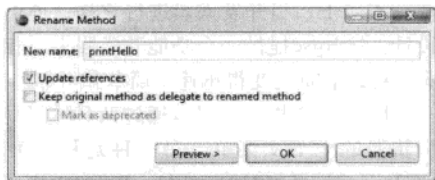


图1-48 重命名方法对话框

清理向导 Eclipse有一个清理向导以用于同时修复多个源代码问题（如移除未使用的私有字段和本地变量）。可以通过Source > Clean Up...命令使用该向导。

1.7.4 本地历史记录

每当你文件做出更改并保存时，该文件的一个快照在同一时间被保持至Eclipse本地历史记录中。这样提供了一种将文件回滚至较早版本以及比较当前版本和较早版本的区别的方法。本地历史记录的每一个条目由它被创建的日期和时间所标识。

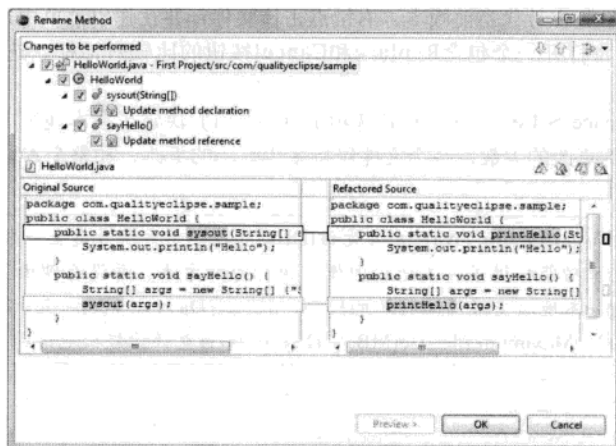


图1-49 重命名方法预览

请注意本地历史记录确实只存在于本地计算机。它不会存储于CVS或其他源代码库中。这意味着本地历史记录只对你可用，而不对其他人可用。这对于VisualAge for Java或ENVY用户是令人惊讶的。这些用户会期待在代码库中可用“方法条件”。

外部文件警告 本地历史仅存储你工作区内的文件的历史记录。如果你使用Eclipse编辑外部文件，不会存储相应的历史记录。

为了将文件的当前状态与一个较早版本相比较，可以右键点击该文件并在上下文菜单中选择 Compare With > Local History...命令。这将打开历史视图，以显示该文件的历史记录。在历史记录列表中双击任意项目将可以看到与该文件的当前状态的比较（图1-50）。

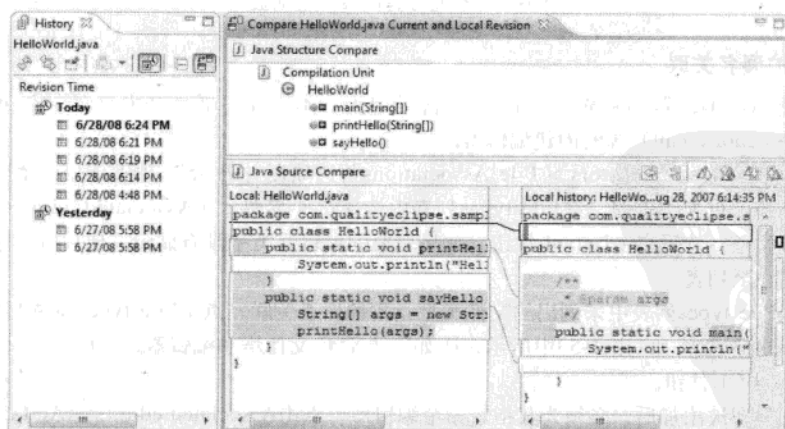


图1-50 与本地历史记录相比较的对话框

为了用文件的较早版本替代当前版本，右键点击该文件，并在弹出菜单中选择Replace With > Local History...。这将打开一个包含Replace和Cancel按钮的比较对话框。选择一个版本并点击Replace按钮。

General > Workspace > Local History首选项页（图1-51）决定了在本地历史记录中存储多少信息。你可以控制保持的更改的天数，每个文件保持多少唯一的更改，和整个本地历史记录文件的最大文件大小。

提示 许多以前的VisualAge for Java用户喜爱该IDE的转换任意方法或类至一个较早版本的功能。Eclipse本地历史记录功能提供了在本地范围模拟该行为的方法。并没有理由（而不是因为磁盘空间）以保持Eclipse本地历史设置为默认的较小的值。将Days to keep files设置为365，Entries per file设置为10 000，Maximum file size(MB)字段设为100将允许你轻松地访问一整年的更改。

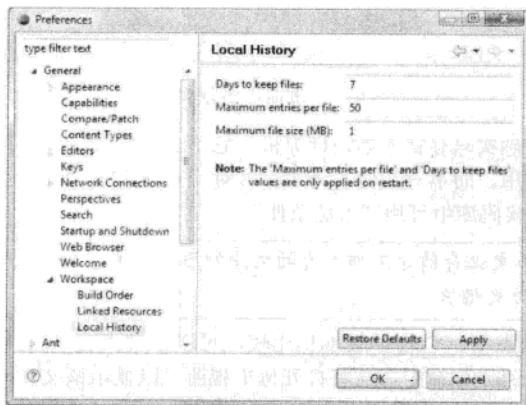


图1-51 本地历史记录首选项页

1.7.5 文件扩展名关联

作为内建Java编辑器的补充，Eclipse包含了用于文本文件、插件开发文件（如plugin.xml、fragment.xml和feature.xml）等的内建编辑器。

你可以通过General > Editors > File Associations首选项页（图1-52）将编辑器分配给一个特定的文件类型。为了更换编辑器，在File types列表中选择文件类型，在Associated editors列表中选择想要的编辑器类型，然后点击Default按钮。如果想要的编辑器类型没有显示，使用File types > Add按钮来将它添加至列表。

为了给在File types列表中未列出的文件类型指定编辑器关联，点击File types > Add按钮以打开New File Type对话框，如同图1-53中所示。比如，为XML文件增加编辑器，在File type字段中输入“*.xml”，并点击OK按钮。

当新文件类型被添加后，必须为其指定一个编辑器。点击Associated editors > Add...按钮以打开Editor Selection对话框。默认地，不同的内建编辑器类型将会在编辑器列表中呈现。

可以通过选择External Programs单选按钮（图1-52）来查看可用的外部编辑器列表。如果你

的系统中装有一个XML编辑器（如Dreamweaver），从列表中选中它并点击OK按钮。该编辑器将会添加至Associated editors列表并自动成为默认XML编辑器（前提是没有其他默认编辑器）。

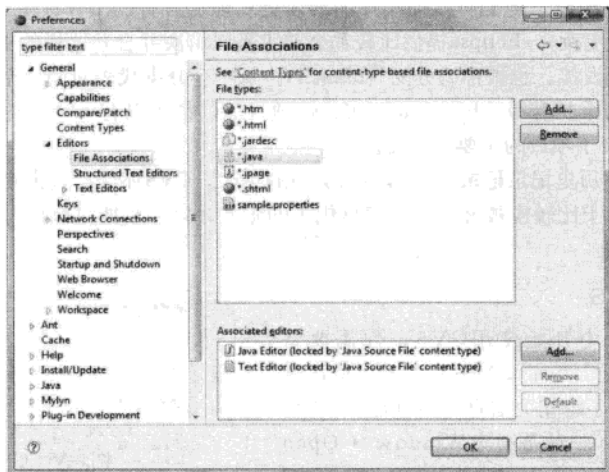


图1-52 文件关联选项页

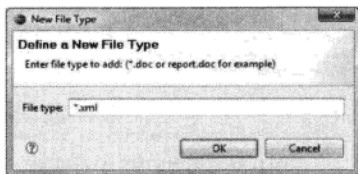


图1-53 新建文件类型对话框

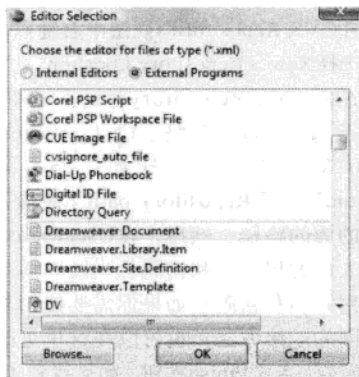


图1-54 编辑器选择对话框

提示 如果你需要经常编辑XML文件，那么来源于www.xmlbuddy.com的XMLBuddy插件是可以集成于Eclipse的几个XML编辑器之一（见附录A）。XMLBuddy编辑器提供了用户定义的语法高亮显示、文档类型定义（DTD）驱动的代码辅助、XML合法性检查以及许多其他功能。

1.8 使用CVS进行团队开发

一般地，你可能要作为一个团队的成员进行开发，并与团队其他成员共享你的代码。本节展示如果设置和使用Eclipse包含的CVS。

团队成员为项目的不同方面工作，在他们各自的工作区内会进行各种更改。当他们准备与团队其他成员共享这些更改时，他们可以将更改提交至共享的CVS库。同样地，当他们需要获取团队其他成员做出的任意更改时，他们可以使用库的内容更新他们各自的工作区。当发生冲突时（比如，对相同资源做出不同更改），Eclipse提供比较和合并工具以解决并合并这样的更改。

CVS提供多个开发流，也被称为分支（branch）。每一个分支代表对同一个组资源的一个独立的更改集。对于不同的维护升级、bug修复、实验性项目等，都可以有多个并行分支。主分支，也被称为“HEAD”，代表了项目的主要工作流。

如同Eclipse本地历史记录记录过去对不同资源的更改，CVS库保持过去对所有资源的所有已提交更改。资源可以用于比较或被替代为任意更优先的版本，这一功能可以由与本地历史记录中所使用的类似工具来实现。

1.8.1 开始使用CVS

为了在Eclipse中开始使用CVS，你需要将Eclipse工作区连接至CVS库（关于设置库，请访问www.cvshome.org以了解相关信息）。可以通过两种方法开始使用CVS：一种是通过Window > Open Perspective > Other...命令打开CVS Repository Exploring透视图；另一种是通过使用Window > Show View > Other...命令打开CVS Repositories视图。然后，在CVS Repositories视图中右键单击，在弹出菜单中选择New > Repository Location...（图1-55）。

在Add CVS Repository对话框（图1-56）中，你需要指定库的位置和登录信息。在Host字段中输入你的CVS主机地址（比如，“cvs.qualityeclipse.com”），在Repository path字段输入相对于主机地址的库的路径。然后，在User和Password字段分别输入你的用户名和密码。当匿名访问时，可以不输入用户名和密码。如果你需要使用与默认连接类型不同的连接方法，也可以在这里更改。当这些工作完成后，点击Finish按钮。如果找到了该CVS库，那它将出现在CVS Repositories视图中。

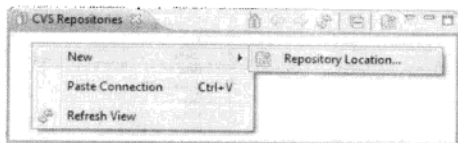


图1-55 CVS库视图

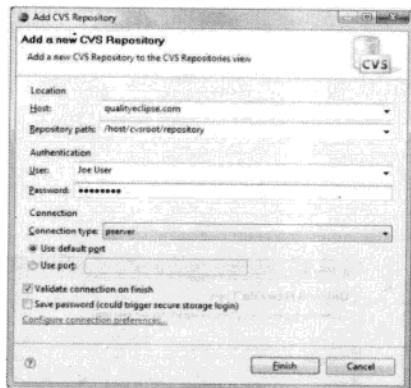


图1-56 添加CVS库对话框

提示 New Project向导也提供了一个创建新的库的选项。选择New > Project...然后选择CVS > CVS Repository Location向导。该向导的第一页与图1-56所示的对话框类似。

1.8.2 从CVS中导出项目

为了从CVS库中导出项目，展开库的位置，然后扩展HEAD项，直到看到你想要导出的项目。右键单击该项目，在弹出菜单中选择Check Out（图1-57）。这将把该项目导入到工作区。

为了将项目导出至特定的项目（比如，工作区外部的项目），可以使用Check Out As...命令。

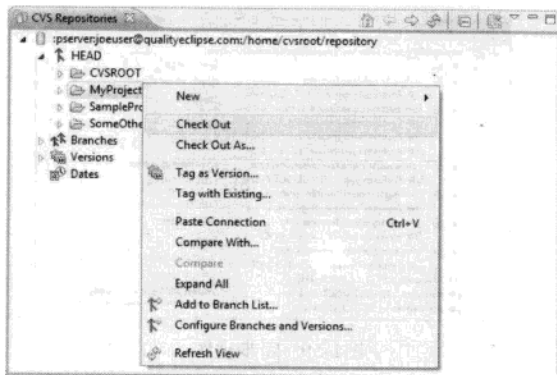


图1-57 导出项目

提示 New Project向导也提供了从一个已有的库中导出项目的选项。选择New > Project..., 然后选择CVS > Projects from CVS向导。在第一页, 选择Use existing repository location选项, 然后选择你准备链接的库, 然后点击Next。在向导的第二页选择Use an existing module。这样将呈现该库中的项目的列表。选择你想要的项目, 并点击Finish按钮。

1.8.3 与库同步

一旦项目中资源被更改, 那这些更改就应该提交回库。右键点击资源 (或是包含该资源的项目), 并选择Team > Synchronize with Repository (图1-58)。

在将工作区的资源与库中的对应资源进行比较后, 将打开Synchronize视图 (图1-59)。Incoming Mode图标将使视图仅显示导入的更改, 而Outgoing Mode图标将使视图仅显示导出的更改 (Incoming/Outgoing模式是第三个选项)。

右键点击导出更改, 并选择Commit...命令以将这些更改提交至库。右键点击导入更改, 并选择Update命令将把这些更改导出至工作区。

如果有冲突发生 (比如, 你和其他开发者对同一资源做出更改), 你将需要使用同步视图中提供的合并工具解决冲突, 然后将合并后的版本提交至库。

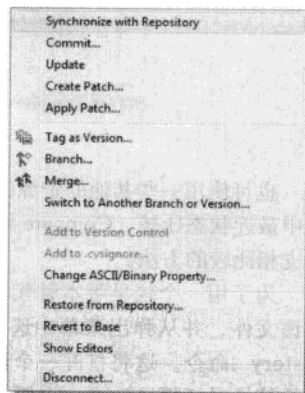


图1-58 团队上下文菜单

1.8.4 比较与替代资源

为了将文件的当前状态与库中一个较早版本相比较, 右键点击该文件并在弹出菜单中选择Compare With > History...命令。这样将打开History视图, 其中显示了根据HEAD分支或其他任意分支所产生的该文件的较早版本 (图1-60)。双击版本列表的任意项目, 将看到与该文件的当前版本的比较。

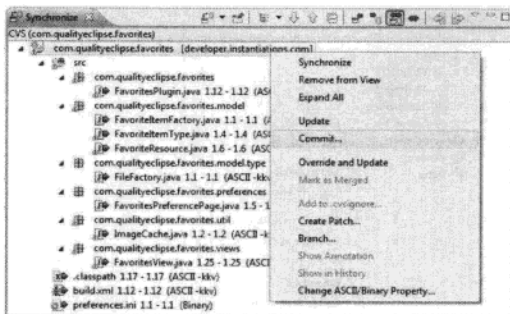


图1-59 同步视图

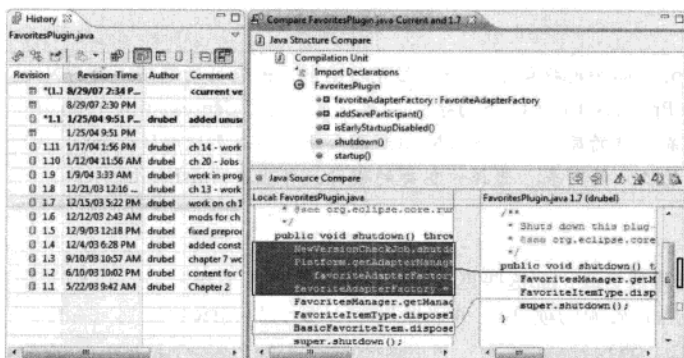


图1-60 版本比较编辑器

也可使用一些其他的资源比较命令。Compare With > Latest from Head命令将资源的当前状态与库中最近状态比较，Compare With > Another Branch or Version命令提供了将一个资源与特定版本或分支相比较的方法。

为了用一个较早版本替代当前版本，右键点击该文件，并从弹出菜单中选择Replace With > History...命令。这将打开一个具有替换和取消按钮的替换对话框。选择一个版本并点击替换按钮以将该版本导入工作区。

1.8.5 CVS标签装饰器

为了更容易地查看哪些资源受库的控制和哪些已经做出更改但没有提交，CVS提供了一系列标签装饰以改进图标与CVS控制的资源。为了打开CVS标签装饰器，使用General > Appearance > Label Decorations首选项页（图1-61）。

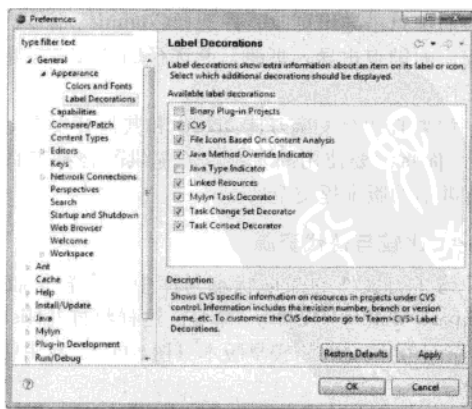


图1-61 标签装饰首选项页


实际被添加的装饰由Team > CVS > Label Decorations首选项页控制。默认地，导出的更改有“>”前缀。

提示 为了让导出的更改更容易使用，打开General > Appearance > Colors and Font首选项页，并将CVS > Outgoing Change (Foreground)设置为蓝色。这将修改所有已被更改待提交至库的资源的前景颜色。

1.9 运行程序

任意包含main()方法的Java应用程序，包括在1.4.4节使用Java类向导创建并在1.7.2节进行更改的.java文件，有一个可运行图标装饰（一个小绿色三角），表示它是可运行的。本节展示了启动（运行）一个Java应用程序的不同方法。

1.9.1 启动Java程序

最简单的运行Java程序的方法是选中该类，然后选择运行菜单中的Run As > Java Application命令（Ctrl+Shift+X, J）或点击工具栏的 运行按钮（图1-62）。这将执行该程序的main()方法，并将所有输出（以黑色字）和错误文本（以红色字）输出至Console视图（图1-63）。

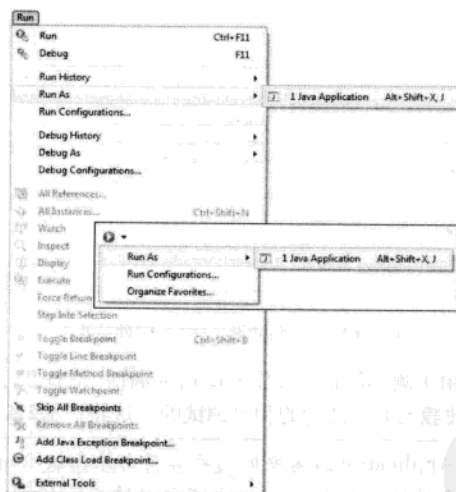


图1-62 Run As > Java Application命令

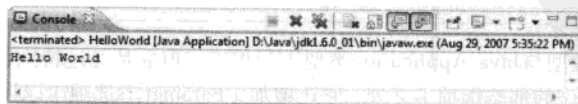




图1-63 控制台视图

一旦应用程序已经运行，它可以通过Run > Run History菜单或 Run工具栏按钮再次运行。点击工具栏的 Run工具栏按钮或按Ctrl+F11将再次运行最近一次运行的程序。

1.9.2 启动配置

当你第一次使用Run As > Java Application命令运行程序时，将创建一个启动配置（launch configuration）。启动配置记录了启动特定Java程序所需要的信息。除了指定Java类的名称之外，启动配置也可设置程序和虚拟机（Virtual Machine, VM）参数、JRE和classpath的使用，甚至包括当运行或调试程序所使用的默认透视图。

启动配置可以通过Run > Run...（图1-65）命令打开启动配置对话框进行编辑。Main选项卡指定所要运行的项目和类，Arguments选项卡记录程序参数（由空格隔开的字符串）和虚拟机参数；JRE选项卡指定所使用的JRE（默认为在Java > Installed JREs首选项中设置的JRE），Classpath选项卡用于覆盖或扩展查找运行项目所需类文件的默认路径；Source选项卡指定源文件的位置，以在调试时显示项目的源代码。Environment选项卡用于设置环境标量；Common选项卡记录启动配置存储位置 and 标准输入和输出应指向的位置的信息。



图1-64 运行历史记录

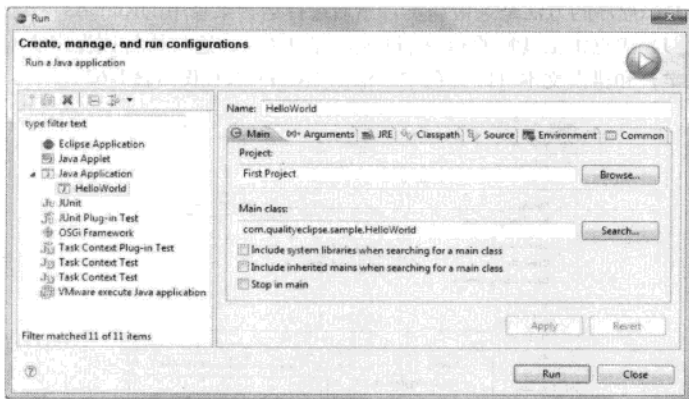


图1-65 启动配置（运行）对话框

Eclipse Application配置用于测试你正在开发的Eclipse插件。它提供了启动另一个工作台的机制。该工作台决定了哪些插件是被载入的、活动的和可调试的。这将在第2章进行更详细讨论。

提示 启动一个Eclipse Application所需要的内存容量由你在该Eclipse中所载入的内容决定。你可以考虑增加给这次启动所分配的内容，这可以通过在Eclipse Application启动配置的Arguments选项卡进行设置。在参数字段可以输入和以下类似的内容：

```
-Xms40m -Xmx256m -XX:MaxPermSize=256m
```

Java Applet配置类型与Java Application类型十分相似，但它是经过特别设计以使用Java Applet的。除了适用于Java程序的那些选项卡之外，它还增加了Parameters选项卡以设置Applet特定的信息，比如宽度、高度、名称和applet参数。

JUnit配置用于运行JUnit测试用例。JUnit测试用例（JUnit test case）是一种特殊的Java程序。大部分配置选项与Java程序和applet的类似。它还增加了一个Test选项卡以用于指定将要执行的测试


用例的特定参数。JUnit将在1.11节进行更详细讨论。

JUnit Plug-in Test配置用于给Eclipse插件进行JUnit测试。它与Eclipse Application配置类似，并多了用于JUnit设置的Test选项卡。

提示 当使用JDK1.4或更高版本运行程序时，Eclipse支持调试过程中的代码热替换。如果一个JDK1.4兼容的JRE不是你的默认JRE，你可以通过在启动配置的JRE选项卡的下拉列表中选择该程序以运行或调试。如果列表中没有任何JRE，你可以通过New按钮添加一个。

1.10 调试简介

上一节介绍了如何使用Run菜单下的选项来运行Java程序。所有输出和错误信息都定向至Console视图。在你代码的合适位置放置System.out.println()语句将带来有限的调试能力。幸运地，Eclipse通过所继承的Java调试器提供一个更有效的调试方案。

最简单的调试Java程序的方法是选择该类，然后选择Run > Debug As > Java Application命令 (Ctrl+Shift+D, J)，或点击工具栏的  Debug调试按钮。这将打开Debug透视图 (图1-8)，以用于在代码中步进，设置断点和监视与更改独立变量的值。当你在调试器中第一次运行程序之后，你可以使用Run > Debug History列表来快速重新运行。也可以通过工具栏调试按钮的下拉菜单访问该列表。

1.10.1 设置断点

为了在代码的特定位置停止调试器，可以设置断点。在你打算设置断点的位置，右键点击编辑器的标记栏，并选择Toggle Breakpoint (图1-66) 命令。除了右键点击之外，你还可以双击标记栏以在你打算放置断点的代码行的左侧设置断点。断点标记将出现在指定的源代码行的左侧。

当设置了一个或多个断点以后，程序遇到断点后，将在执行断点所标记代码行之前停下来。调试器将会显示程序的哪些线程在运行，哪些线程被中止。它还显示执行在哪一行代码停止并在编辑器中将那一行设为高亮 (图1-67)。

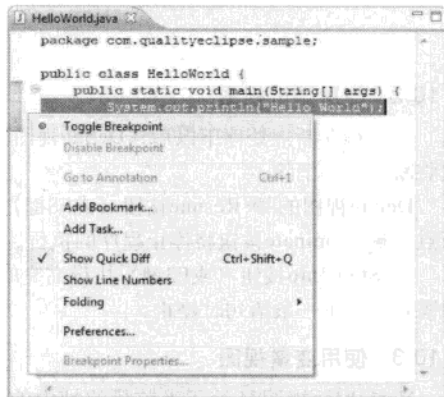


图1-66 添加断点

提示 如果你遇到一个不是你设置的断点，请仔细查看该断点是如何产生的。对于一个可用的断点而言，你将会看到一个蓝色的圆点或带有标记的蓝色圆点。检查标记图标在启动虚拟机并载入类中存在该断点的情况下才会出现。

提示 在Run/Debug > Launching首选项页选中Remove terminated launches when a new launch is created单选框以自动清理较老的启动。

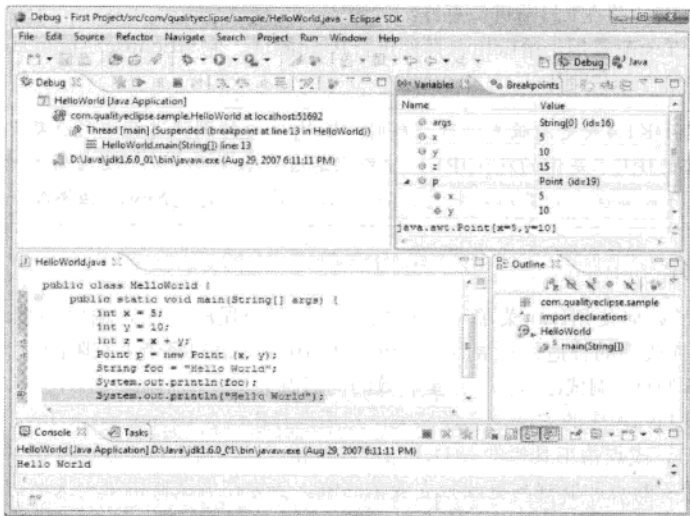


图1-67 停止于断点的调试器

1.10.2 使用调试视图

当执行在断点处停下以后，Debug视图提供了几种恢复执行的方法，在程序内部进行语句步进，或将断点全部消除。

Debug视图的 Resume按钮（或F8键）将会恢复程序的执行，直到程序自己停止或遇到另一个断点。 Terminate按钮将停止程序的执行。

Step Into按钮（或F3键）执行高亮的下一条语句， Step Over按钮（或F6键）将跳过高亮的语句，在下一条语句时停止。

1.10.3 使用变量视图

Variables视图显示了当前栈的帧中的变量的状态（图1-68）。选择一个变量将会在视图底部的细节栏显示它的值。基本类型变量将直接显示它们的值，而对象类型可以扩展以显示它们的独立元素。你可以在该视图中更改基本变量的值，但你不能改变对象类型的值，除非你使用Expressions视图（参见1.10.4节）。请注意，当你在程序中步进时，Variables视图中列出的变量的值会随之更改。

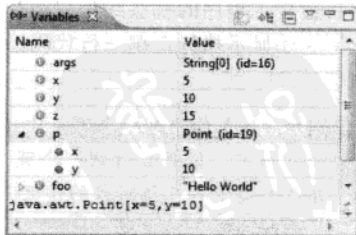


图1-68 变量视图

1.10.4 使用表达式视图

Expressions视图（图1-69）提供了监视调试器中的值的方法，它还可以用来查看输入编辑器的不同表达式的结果、Variables视图的细节栏和Expressions视图的细节栏。

为了使用Expressions视图，首先选择一个表达式执行。这条表达式既可以是已有的，也可以是

用户输入的。然后，在Variables视图或Expressions视图编辑器的弹出菜单中选择Watch、Display或Inspect命令。

如果你在编辑器中选定表达式并选择Display命令时，结果将出现于Display视图中。当在Variables视图或Expressions视图中选择一个表达式时，结果将会在那个视图的细节栏出现。

如果你选择Inspect命令，一个包含表达式结果的窗口将弹出。按下Ctrl+Shift+I将把结果移至Expressions视图。如Variables视图一样，基本类型变量直接显示值，而对象类型变量将可以扩展以显示它们的独立元素。

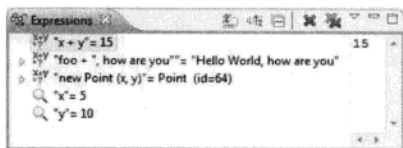


图1-69 表达式视图

1.11 测试简介

除了通过运行或调试来手动测试程序之外，Eclipse还支持JUnit框架（参见www.junit.org）以用来创建和运行可重复的测试用例。

1.11.1 创建测试用例

为了创建测试用例，首先你必须将junit.jar文件（来源于org.junit插件）作为库添加至你的项目，这可以通过使用Java Build Path > Libraries项目属性页实现。点击Add Library...按钮，选择“JUnit”，点击Next，选择“JUnit 3”，然后点击Finish。

当以上步骤完成后，选择你想要创建测试用例的类，打开New向导，选择Java > JUnit > JUnit Test Case选项。这将激活JUnit Test Case向导（图1-70），以用于创建JUnit测试用例。如果你忘记将junit.jar添加至你的项目，向导将会自动提示你这项工作。

默认地，新建的测试用例的名称是被测试类的名称在后面加上“Test”而成。可供选择地，你可以让向导创建main()、setup()和teardown()方法以及被测试类的所有public和protected方法创建测试方法。

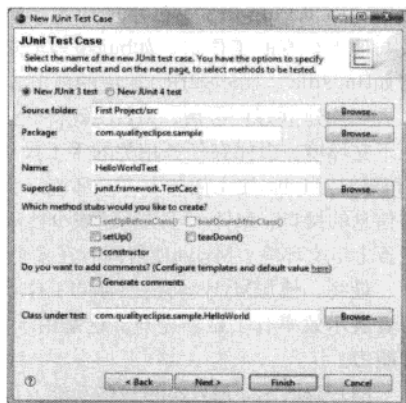


图1-70 JUnit测试用例向导

提示 CodePro包含一个更高级的Test Case向导，它提供了对Eclipse测试用例向导的多项增强功能，如指定任意测试用例，生成更好的默认代码和对固定测试创建的支持。

1.11.2 运行测试用例

测试用例创建好之后，选择测试用例类（或包含测试用例的项目和包），并选择Run > Run As > JUnit Test命令（Alt+Shift+X, T）。这将打开JUnit视图（图1-71）以展示测试的运行结果。Failure选项卡显示了该测试用例的所有已记录的失败，Hierarchy选项卡以树形结构显示整个测试包。

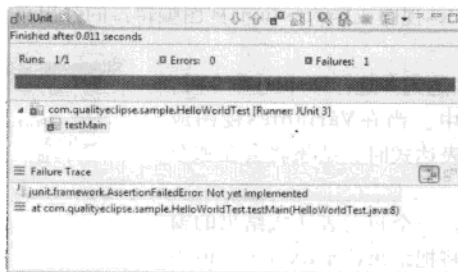



图1-71 JUnit视图

如果有测试失败，更正问题并点击JUnit视图中的  Rerun Test按钮重新运行测试。或者，从选定测试用例并从Run菜单或工具栏按钮重新运行。如果你需要自定义测试设置，选择Run > Run...命令以打开启动配置对话框（图1-65）。

1.12 Mylyn简介

Mylyn是几个Eclipse版本的内置组件。这些版本包括Java Developers版本、Java EE Developers版本和RCP/Plug-in Developers版本。Mylyn提供了一种强大的管理任务的功能。任务是你想要记忆或与他人分享的工作集，如bug报告或功能请求。Mylyn允许你在工作区本地存储任务或处理存储在诸如Bugzilla、Trac或JIRA库中的任务。你需要对应库的Mylyn连接。

当你建立好任务后，Mylyn就将监视你的活动以确认当前任务的相关信息。Mylyn创建一个任务上下文环境，以保存任务相关的手工操作集（包含你已打开的文件、你已编辑过的方法、你已引用的API等）。Mylyn使用任务上下文环境以将Eclipse用户界面仅仅聚焦于相关信息。通过提供快速访问所需信息的接口，Mylyn可以通过减少搜索、滚动和其他导航操作所需时间以提高工作效率。通过固定任务上下文环境，Mylyn使得同时在多个任务并行工作，返回较高级任务和与他人共享任务变得简单。

最终，使用Mylyn将会使你的工作产生一个细微但重要的改变。Mylyn的关注于任务的界面的支持者表示效率有了显著提升。这是由于它简单管理多个、协作的任务能力和跟踪所有这些任务进度的能力。

图1-72展示了Mylyn的一些功能。比如，Task List（展示单个活动任务和不同的Bugzilla报告）、富任务编辑器（展示了bug属性、描述、注释等）和在Eclipse Package Explorer内部的聚焦于任务的过滤器。

Mylyn提供了便利的界面以用于查询Eclipse Bugzilla跟踪系统（参见21.2.2节）。这将使得查找已有的bug和报告新bug变得容易。

为了在Bugzilla中查找bug或功能请求，打开New向导，选择Mylyn > Query选项，并选择Eclipse.org库和Create query using form选项（图1-73）。

这将打开New Bugzilla Query向导（图1-74），在该向导中你可以输入不同的查询条件。从输入查询标题开始，该标题将出现在Mylyn Task List视图中。然后，选择想要驱动搜索的条件。你可以查询特定的bug编号，在bug概述和评论中查找特定的词语，将查询限制于产品、组件、版本、状态、方案等，或是查询与bug相关的特定人员，如作者、报告者或评论者。点击Finish按钮以锁定查询并将其添加至Task List视图。

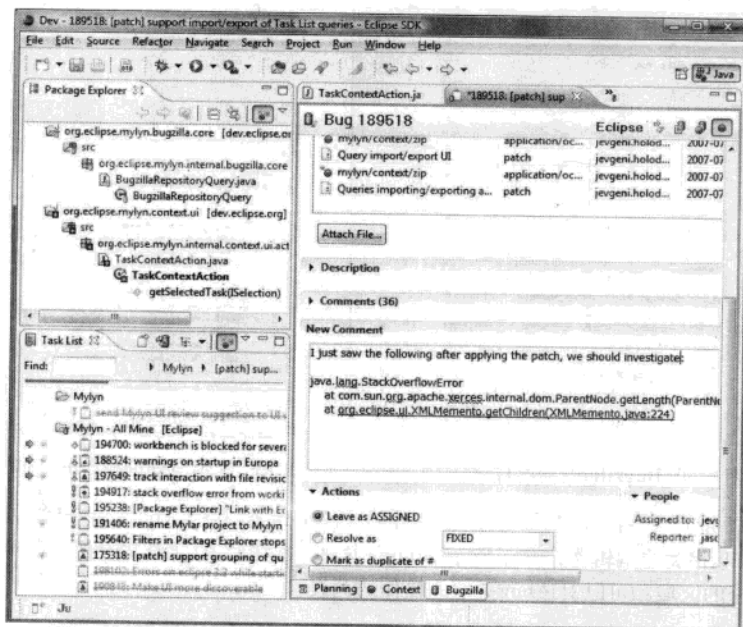


图1-72 Mylyn视图和编辑器

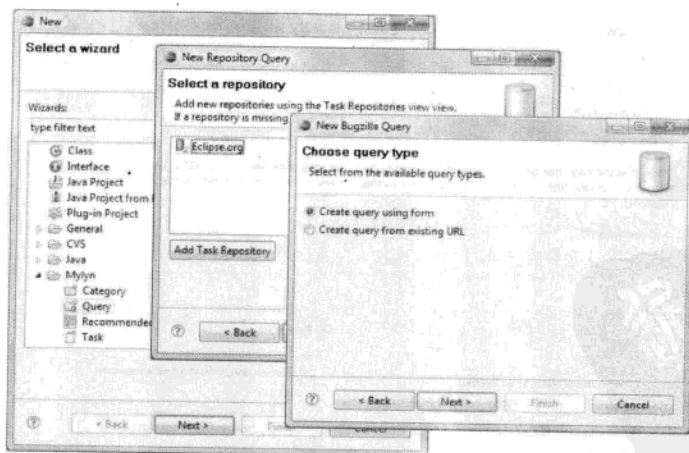


图1-73 新建Bugzilla查询向导

Mylyn Task List视图(图1-75)显示你已创建的查询项目列表。可以扩展任意的查询出口以查看相匹配的Bugzilla条目。用于装饰每一行选项的图标与字体表示它的状态、解决方案和严重程度。

1.13 总结

本章带领你快速浏览了Eclipse IDE的主要组件, 这些组件可以用于Eclipse插件开发。此时, 你应该可以轻松地在Eclipse用户界面上浏览, 并使用内建工具来创建、编辑、运行、调试和测试Java代码。

下一章将着手开始创建第一个Eclipse插件。后续的每一章都会给该插件添加越来越多的细节, 并逐渐将该插件由一个简单的示例转变为一个可以用于Eclipse日常开发的具有强大功能的工具。

参考文献

Eclipse-Overview.pdf (available from the eclipse.org Web site).

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer's Guide to Eclipse, Second Edition*. Addison-Wesley, Boston, 2004.

Arthorne, John, and Chris Laffra, *Official Eclipse 3.0 FAQs*. Addison-Wesley, Boston, 2004.

Carlson, David, *Eclipse Distilled*. Addison-Wesley, Boston, 2005. Eclipse Wiki (see wiki.eclipse.org).

CVS (see www.cvshome.org).

CVS Howto (see wiki.eclipse.org/index.php/CVS_Howto).

Mylyn (see www.eclipse.org/mylyn).

Mylyn Tutorial (see www.ibm.com/developerworks/java/library/j-mylyn1).

Mylyn Wiji (see wiki.eclipse.org/index.php/Mylyn).

Fowler, Martin, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, 1999 (www.refactoring.com).

Glezen, Paul, "Branching with Eclipse and CVS." IBM, July 3, 2003 (www.eclipse.org/articles/Article-CVS-branching/eclipse_branch.html).

JUnit (see www.junit.org).



第2章 简单插件示例

在讨论Eclipse基础结构（第3章）和深入探讨插件开发的具体内容之前，创建一个用于讨论和示例的简单插件是十分有用的。本章将带领你一步步创建一个简单但完全可操作的示例插件。我们将在本书剩下的内容中不断完善其功能。该过程提供了关于使用Eclipse IDE以及创建和维护插件的所有方面的有价值的信息。

2.1 收藏夹插件

你将在本书中创建并使用的Favorites插件显示了一个资源列表，并让你可以从列表中添加和删除资源，在选定资源上轻松打开一个编辑器，根据系统事件自动升级列表等。后续章节讨论改进Favorites插件的插件开发的不同方面。

本章首先创建收藏夹插件的简易样式，主要包括以下步骤：

- 创建插件项目
- 评审生成代码
- 构建产品
- 安装并运行产品

2.2 创建插件项目

第一步是使用Eclipse New Project向导创建一个插件项目。除了创建新项目之外，该向导还具有一些不同的代码生成选项，如视图、编辑器和操作等，以用于创建简单的插件代码。为了操作简便和仅仅关注插件创建的基本方面，选择Plug-in with a view选项。这将在本节进行讨论。

2.2.1 新建插件项目向导

从File菜单选择New > Project以启动New Project向导（图2-1）。在向导的第一个页面，在列表中选择Plug-in Project，然后点击Next按钮。

在向导的下一页（图2-2），输入项目名称，在本章的示例中，请输入com.qualityeclipse.favorites，这也将作为收藏夹插件的标识符。第3章讨论了插件标识符和插件结构的其他方面的更详细内容。按照图中所示填好其余内容，并点击Next按钮。

提示 可以任意命名项目，但如果将它用插件标识符命名，那很多事都将变得更为容易。作为约定，这是Eclipse组织用于其大部分工作的插件项目的命名方案。基于这一点，新建项目向导假设项目名称和插件标识符是一致的。

2.2.2 定义插件

每一个插件都具有一个META-INF/MANIFEST.MF文件。此外，它还可能包括一个plugin.xml文件和程序中代表插件的Java类。向导的下一页显示了生成插件清单文件和插件Java类的相关选项。

如图2-3所示，为插件提供Plug-in ID、Plug-in Version、Plugin Name和其他信息。然后点击Next按钮。

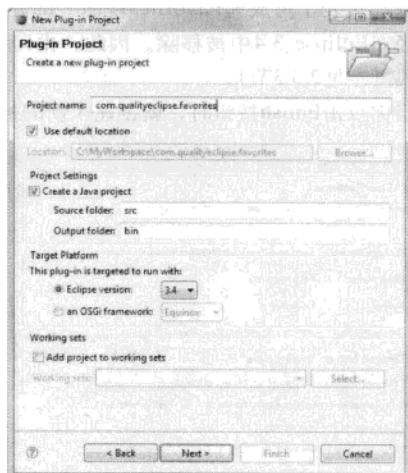
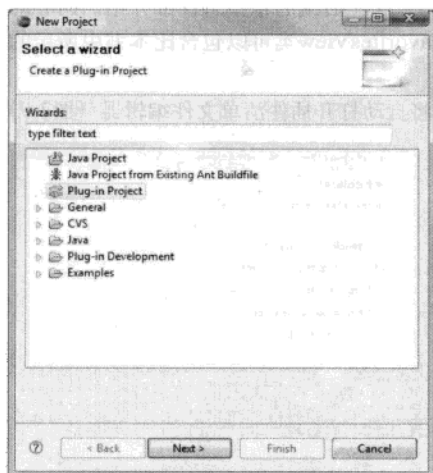


图2-1 新建项目向导第一个页面——选择项目类型 图2-2 新建项目向导第二个页面——给项目命名

然后，New Plug-in Project向导的下一页面显示了可以由向导自动生成的不同的插件片段（图2-4）。在这一页面上，有许多用于生成示例代码的不同选项。尝试每一个选项，并查看所生成的代码是非常有益的。对本项目来说，选择Plug-in with a view并点击Next按钮。

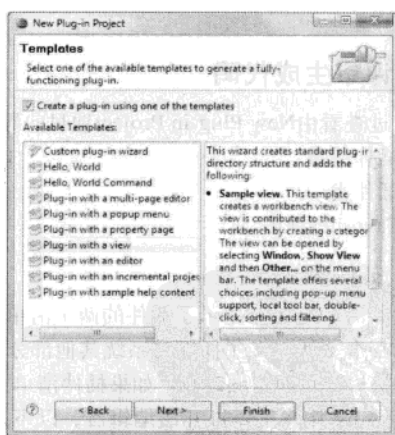


图2-3 新建项目向导第3页——描述插件

图2-4 新插件项目向导第4个页面——选择插件类型

2.2.3 定义视图

这一过程中的下一步工作是选择视图代码生成选项。在该页面上输入值（图2-5），不选中Add the view to the resource perspective和Add context help to the view（仅适用于Eclipse 3.4）两个单选框，以简化所生成的插件清单文件。

如果你使用的是Eclipse 3.3, 点击Next按钮并取消选中所有的代码生成选项(图2-6)。每一个单选框都代表了可成为Favorites视图一部分的生成代码。这在后续的章中有详细介绍。向导的该页面已经在Eclipse 3.4中被移除。因此, 由向导生成的FavoritesView类可以包含比本书中所示的更多的代码(参见2.3.3节)。

当你点击Finish按钮时, 就创建了新的插件项目。将自动打开插件清单文件编辑器(图2-9)。

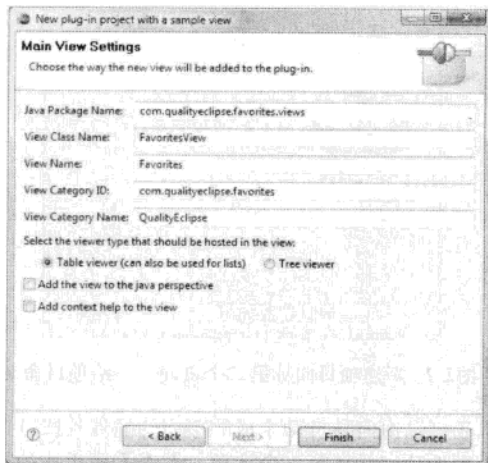


图2-5 新建插件项目第5页面——定义视图

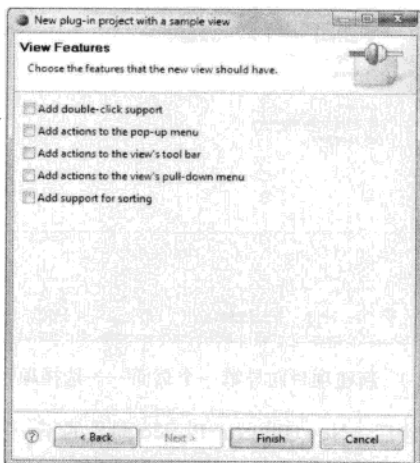


图2-6 新建插件项目第6页面——视图的代码生成选项 (仅适用于Eclipse 3.3)

2.3 评审生成代码

通过查看由New Plug-in Project项目向导生成的代码展示了示例插件的以下主要部分:

- 插件清单
- 插件类
- Favorites视图

2.3.1 插件清单

插件清单编辑器显示了插件的两个清单文件——META-INF/MANIFEST.MF 和plugin.xml的内容。这两个文件定义了插件与系统其他部分的关系。创建了该新插件项目之后, 该编辑器将自动打开它的第一个页面(图2-9)。如果插件清单文件编辑器被关闭了, 双击META-INF/MANIFEST.MF或plugin.xml文件将重新打开该编辑器。以下是清单文件编辑器的概述, 而插件清单文件的更多细节在第3章。

虽然编辑器是一个便利的修改插件描述的方法, 但查看源代码以了解编辑器的不同部分是如何与底层代码相关的仍然具有一定意义。点击MANIFEST.MF选项卡, 以显示META-INF/MANIFEST.MF文件的源代码。该文件定义了插件的运行时相关内容(图2-7)。头两行定义清单文件是一个OSGi清单文件(参见3.3节)。之后的行定义了插件名称、版本、标识符、classpath和所依赖的插件。所有这些内容都可以在插件清单编辑器的其他页面进行编辑。

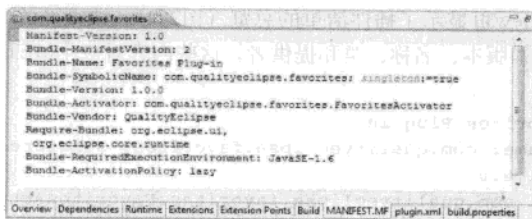


图2-7 插件清单编辑器的MANIFEST.MF页面

Eclipse 3.4新增内容 MANIFEST.MF文件中的Eclipse-LazyStart: true指令已经由Bundle-ActivationPolicy: lazy替代。两条指令具有相同的含义，只是名称变了。

点击编辑器的plugin.xml选项卡将显示plugin.xml文件。该文件定义了插件的扩展内容（图2-8）。第一行声明这是一个XML文件，之后的行定义插件扩展项。

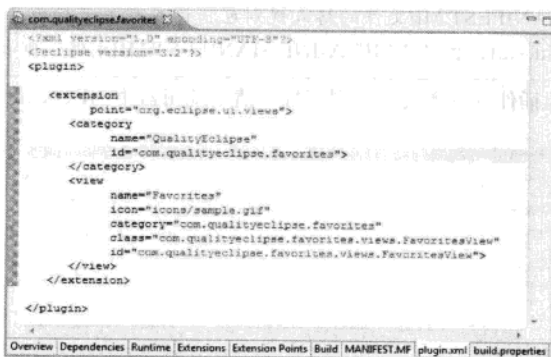


图2-8 插件清单编辑器的plugin.xml页面

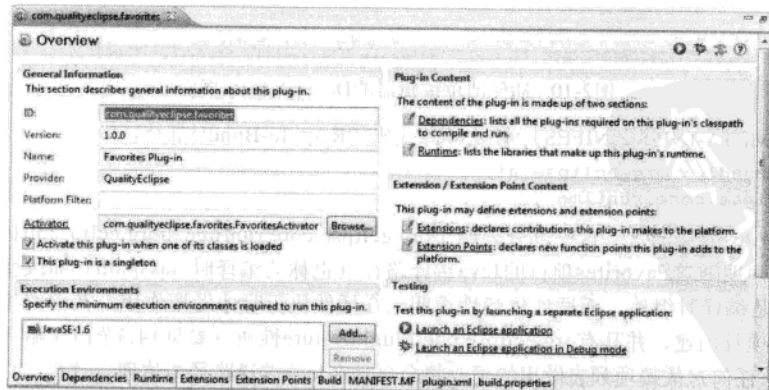


图2-9 插件manifest编辑器的概述页，This plug-in is a singleton选项是Eclipse 3.4新增内容

清单编辑器的Overview页显示了插件清单的总览(图2-9)。该页面的内容描述了插件的一般信息,如插件标识符(ID)、版本、名称、类和提供者,这些内容对应于META-INF/MANIFEST.MF文件的第一部分:

```
Bundle-Name: Favorites Plug-in
Bundle-SymbolicName: com.qualityeclipse.favorites; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.qualityeclipse.favorites.FavoritesActivator
Bundle-Vendor: QualityEclipse
Bundle-RequiredExecutionEnvironment: JavaSE-1.5
```

你可以在Overview页面编辑相关信息,也可以切换至MANIFEST.MF页面以直接编辑源代码。

提示 在除了plugin.xml和MANIFEST.MF页面之外的所有页面做出更改都将导致清单编辑器重新格式化源代码。如果你对于清单文件的格式有特殊要求,那只有两种选择:一是只使用plugin.xml和MANIFEST.MF页面来进行编辑;二是使用另一种编辑器。

警告 META-INF/MANIFEST.MF文件的格式规则包括许多有关于行长度和行包裹的非直观规则。请小心编辑plugin.xml,但对于META-INF/MANIFEST.MF文件,则要慎重!

本插件与系统中其他插件的依赖关系出现于插件清单编辑器中的Dependencies页面(图2-10)。

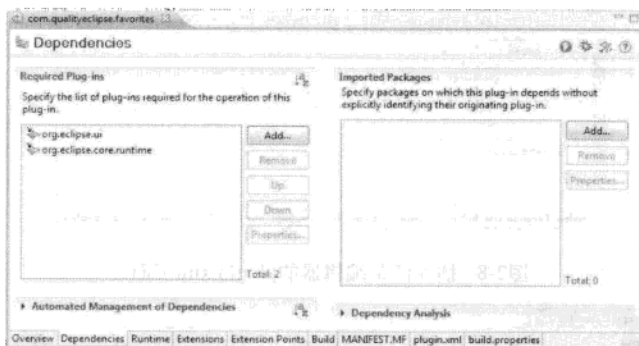


图2-10 插件清单编辑器的Dependencies页面

这对应于META-INF/MANIFEST.MF文件源代码的Require-Bundle部分:

```
Require-Bundle: org.eclipse.ui,
                org.eclipse.core.runtime
```

对于Favorites插件而言,这一部分表明了org.eclipse.core.runtime和org.eclipse.ui插件的依赖关系。该依赖性声明随着Favorites项目的Java编译路径(也称为编译时classpath)而改变。这是因为Java编译路径是编译时组件,而插件依赖性声明只在插件执行时才发挥作用。其中的原因在于该项目是作为插件项目创建,并具有org.eclipse.pde.PluginNature性质(参见14.3节以了解更多有关项目特性的信息)。任何对依赖项列表做出的更改将会自动在Java编译路径中体现,这样,你就可以拥有一个可以编译、创建但还不能正确执行的插件了。

提示 请编辑该依赖项列表而不是Java编辑路径，这样这两者可以总是自动保持同步。

此外，依赖项还可能表现为清单编辑器Dependencies页面的Imported Packages（图2-10和3.3.3节的结尾部分）中。这将对应用于META-INF/MANIFEST.MF文件的Import-Package部分，如下所示：

```
Import-Package: org.eclipse.ui.views,
               org.eclipse.core.runtime.model
```

清单编辑器的Runtime页面（图2-11）对应于META-INF/MANIFEST.MF文件的Bundle-ClassPath部分。该部分内容定义了：哪些库将和插件一同传递并被插件在执行时使用；其他插件是否可以引用库中的代码（参见21.2.5节以了解关于包可见性的更多内容）。对于Favorites插件，所有的代码均包含于com.qualityeclipse.favorites_1.0.0.jar它本身，因此不需要Bundle-ClassPath声明。

Favorites插件没有导出任何包以供其他插件使用或扩展。

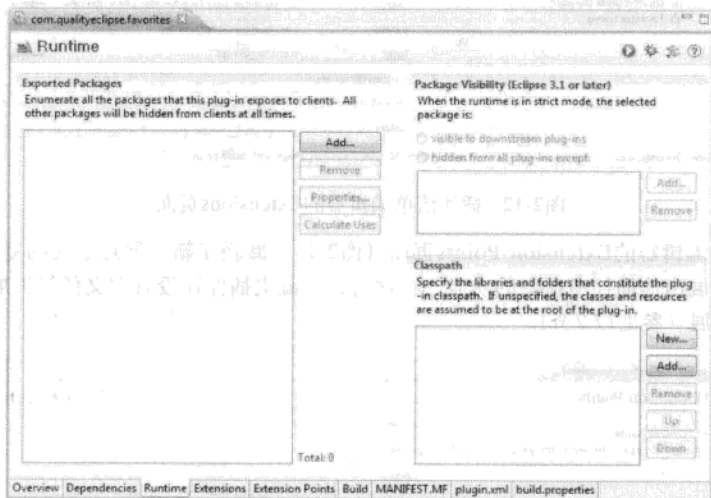


图2-11 插件清单编辑器运行时页面

Extensions页面（图2-12）显示了本插件如何扩展系统其他插件已提供的功能，对应于plugin.xml文件的<extension point="org.eclipse.ui.views">部分：

```
<extension
    point="org.eclipse.ui.views">
  <category
    name="QualityEclipse"
    id="com.qualityeclipse.favorites">
  </category>
  <view
    name="Favorites"
    icon="icons/sample.gif"
    category="com.qualityeclipse.favorites"
    class="com.qualityeclipse.favorites.views.FavoritesView"
    id="com.qualityeclipse.favorites.views.FavoritesView">
```



```
</view>
</extension>
```

Favorites插件通过提供QualityEclipse视图的一个附加类别和Favorites类别的一个新视图声明了使用org.eclipse.ui.views扩展点的org.eclipse.ui插件的一个扩展项。在Extensions页面左侧的树中选择一项目将使该项目的属性出现在右侧。在这里，在Extensions页面中选择Favorites (view)以显示名称、标识符、类和更多Favorites视图已声明的信息。这些内容对应于上面的XML文件的<view>部分所示内容。

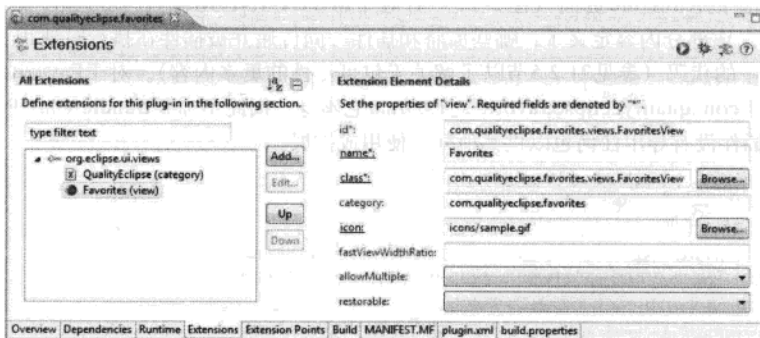


图2-12 插件清单编辑器的Extensions页面

最后，清单编辑器的Extension Points页面（图2-13）减轻了新扩展点定义的复杂程度。因此，其他插件可以扩展由该插件所提供的功能。在这时，收藏夹插件还没有定义任何扩展点，因此还不能被其他插件扩展（参见17.2节）。

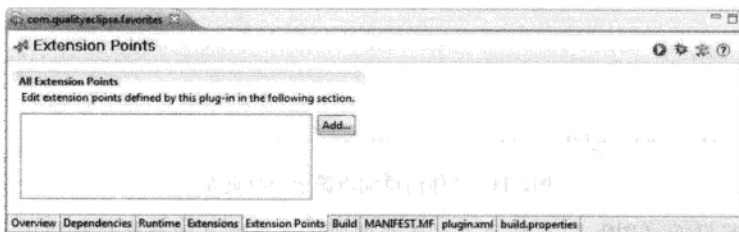


图2-13 插件清单编辑器的Extension Points页面

2.3.2 启动器或插件类

如同清单编辑器中的Overview页面中所示（图2-9），每个插件可以可选地声明一个在程序中代表其自身的类。该类被称为Activator（较早版本的Eclipse称为Plug-in类）。在Favorites插件中，该类被命名为com.qualityeclipse.favorites.FavoritesActivator。

```
package com.qualityeclipse.favorites;

import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;
```

```
/**
 * The activator class controls the plug-in life cycle
 */
public class FavoritesActivator extends AbstractUIPlugin {
    // The plug-in ID
    public static final String PLUGIN_ID
        = "com.qualityeclipse.favorites";

    // The shared instance
    private static FavoritesActivator plugin;

    /**
     * The constructor
     */
    public FavoritesActivator() {
    }

    /**
     * This method is called upon plug-in activation.
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
        plugin = this;
    }

    /**
     * This method is called when the plug-in is stopped.
     */
    public void stop(BundleContext context) throws Exception {
        plugin = null;
        super.stop(context);
    }

    /**
     * Returns the shared instance
     */
    public static FavoritesActivator getDefault() {
        return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path
     *
     * @param path the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path) {
        return imageDescriptorFromPlugin(PLUGIN_ID, path);
    }
}
```

如果META-INF/MANIFEST.MF文件中的Bundle-ActivationPolicy的值是lazy，那么在插件被激活时，在载入其他任何类之前，Eclipse将首先实例化启动类。这对应于清单编辑器Overview页面的

“Activate this plug-in when one of its classes is loaded.” 单选框（图2-9）。该启动类实例被Eclipse用于插件的生存周期中，并且不创建其他实例。

提示 参见http://wiki.eclipse.org/Lazy_Start_Bundles以了解关于Bundle-ActivationPolicy的更多背景知识。

一般地，启动器类声明一个静态字段以引用该个体，以使它可以在插件中可以较容易地供其他内容引用。在这里，Favorites插件定义一个字段名为plugin。该字段在开始方法中被分配，并由getDefault方法访问。

提示 Eclipse一般只初始化活动插件启动类的一个实例。请不要自行创建这种类的实例。

2.3.3 收藏夹视图

除了插件清单和插件类之外，New Plug-in Project向导还为名为Favorites的简单视图生成了代码（以下示例）。此时，视图创建并显示一个简单模型的信息；在后续的章中，该视图将会与收藏夹模型相关联，并将会显示模型中所包含的收藏夹项目的信息。Eclipse 3.4还生成了与本练习无关的代码，因此请调整生成代码以与下面所示一致。

```
package com.qualityeclipse.favorites.views;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.Image;
import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.*;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.SWT;

/**
 * This sample class demonstrates how to plug-in a new workbench
 * view. The view shows data obtained from the model. The sample
 * creates a dummy model on the fly, but a real implementation
 * would connect to the model available either in this or another
 * plug-in (e.g., the workspace). The view is connected to the
 * model using a content provider.
 * <p>
 * The view uses a label provider to define how model objects
 * should be presented in the view. Each view can present the
 * same model objects using different labels and icons, if
 * needed. Alternatively, a single label provider can be shared
 * between views in order to ensure that objects of the same type
 * are presented in the same way everywhere.
 * <p>
 */
public class FavoritesView extends ViewPart {
    private TableViewer viewer;
    /**
     * The content provider class is responsible for providing
```

```
* objects to the view. It can wrap existing objects in
* adapters or simply return objects as-is. These objects may
* be sensitive to the current input of the view, or ignore it
* and always show the same content (Task List, for
* example).
*/

class ViewContentProvider
    implements IStructuredContentProvider
{
    public void inputChanged(
        Viewer v, Object oldInput, Object newInput) {
    }
    public void dispose() {
    }
    public Object[] getElements(Object parent) {
        return new String[] { "One", "Two", "Three" };
    }
}
/*
 * The label provider class is responsible for translating
 * objects into text and images that are displayed
 * in the various cells of the table.
 */

class ViewLabelProvider extends LabelProvider
    implements ITableLabelProvider
{
    public String getColumnText(Object obj, int index) {
        return getText(obj);
    }
    public Image getColumnImage(Object obj, int index) {
        return getImage(obj);
    }
    public Image getImage(Object obj) {
        return PlatformUI.getWorkbench().getSharedImages()
            .getImage(ISharedImages.IMG_OBJ_ELEMENT);
    }
}

/**
 * The constructor.
 */
public FavoritesView() {
}
/**
 * This is a callback that will allow us to create the viewer
 * and initialize it.
 */

public void createPartControl(Composite parent) {
```

```

viewer = new TableViewer(
    parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
viewer.setContentProvider(new ViewContentProvider());
viewer.setLabelProvider(new ViewLabelProvider());
viewer.setInput(getViewSite());
}

/**
 * Passing the focus request to the viewer's control.
 */
public void setFocus() {
    viewer.getControl().setFocus();
}
}

```

2.4 构建产品

创建产品包括将需要被分发的元素打包，这样用户就可以在开发环境中进行安装。你可以通过几种不同的方式创建产品，包括手动创建以及使用Windows批处理脚本、UNIX shell脚本或使用Apache Ant脚本。你可以将最终产品作为一个压缩文件或一个单独的可执行文件分发。基于我们的目的，Favorites插件将会以一个包含源代码的单一压缩zip文件进行分发。

2.4.1 手动构建

手动创建产品包括启动Eclipse Export向导，填充一些字段并点击Finish按钮。选择File > Export命令以启动将要使用的导出向导。在向导的第1页（图2-14），选择Deployable plug-ins and fragments，然后点击Next按钮。

在Export向导的第2页（图2-15），选择要导出的插件，输入包含结果的zip文件的名称，并选择所展示的选项。此外，指定该导出操作将保存为Ant脚本，并存储于com.qualityeclipse.favorites项目的build-favorites.xml文件中。然后，选中Include source code单选框，并点击Finish。

该创建的zip文件包含了一个单独的插件JAR文件（在Eclipse 3.1中，插件可以作为单独JAR文件部署）：

```
plugins/com.qualityeclipse.favorites_1.0.0.jar
```

该插件JAR文件包含了Export向导中指定的插件：

```

com.qualityeclipse.favorites classes
com.qualityeclipse.favorites source files
plugin.xml
icons/sample.gif
META-INF/MANIFEST.MF

```

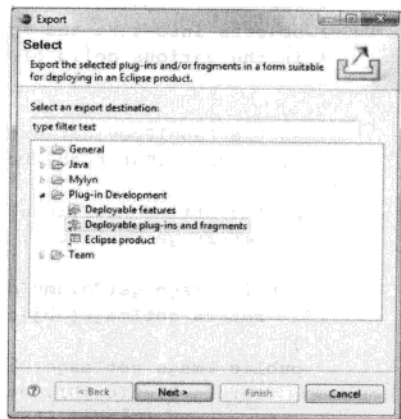


图2-14 导出向导第1页——选择导出类型

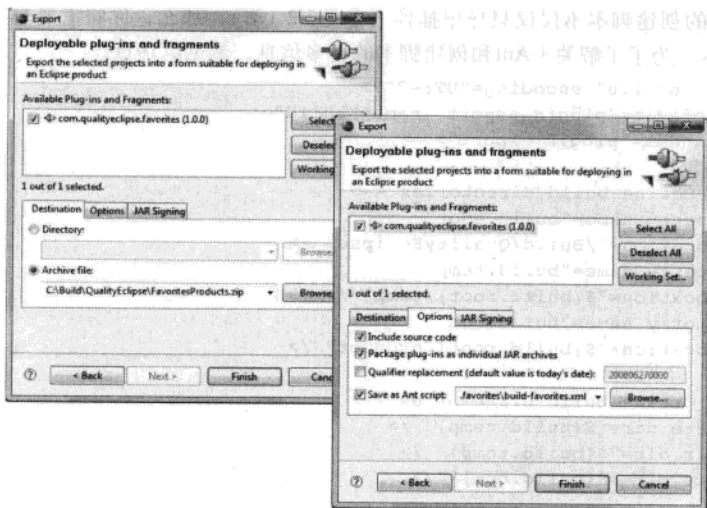


图2-15 导出向导第2页——设定zip文件内容

遗憾的是，该过程是手动的，因此容易产生错误。手动构建产品一两次还好，但如果公司里另一个人需要构建该产品呢？当产品扩展并包含更多插件呢？产品需要一个可重复、可靠的构建方法。

2.4.2 使用Apache Ant构建

Apache Ant脚本为构建插件项目提供了一种可靠、灵活和可重复的方法。建立一个Ant脚本并不需要做太多的前端工作，但是它比起手动构建产品来，出现错误的几率要小得多。为了了解更多有关Ant和创建更复杂构建脚本的信息，参见第19章。

Eclipse可以生成简单的Ant脚本。上一节在导出向导中生成了Ant脚本文件。该脚本文件位于com.qualityeclipse.favorites项目，名为build-favorites.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="plugin_export" name="build">
  <target name="plugin_export">
    <pde.exportPlugins
      destination="C:\Build\QualityEclipse"
      exportSource="true"
      exportType="zip"
      filename="FavoritesProduct.zip"
      plugins="com.qualityeclipse.favorites"
      useJARFormat="true" />
  </target>
</project>
```

该简单的脚本与Eclipse用户界面能良好地吻合，然而遗憾的是，pde.exportPlugins和其他pde.export*任务是异步的，而且不能在无头部的环境中使用（参见Bugzilla条目58413于bugs.eclipse.org/bugs/show_bug.cgi?id=58413），这样使得创建比简单脚本更复杂的脚本就变得十分困难。

如果你想要的创建脚本不仅仅只导出插件（参见3.2.1节），那么，你将需要与下面所示类似的更复杂的Ant脚本。为了了解关于Ant和创建脚本的更多信息，参见第19章。

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="plugin_export" name="build">
  <target name="plugin_export">

    <!-- Define build directories -->
    <property name="build.root"
      location="/Build/QualityEclipse" />
    <property name="build.temp"
      location="${build.root}/temp" />
    <property name="build.out"
      location="${build.root}/product" />

    <!-- Create build directories -->
    <delete dir="${build.temp}" />
    <mkdir dir="${build.temp}" />
    <mkdir dir="${build.out}" />

    <!-- Read the MANIFEST.MF -->
    <copy file="META-INF/MANIFEST.MF" todir="${build.temp}" />
    <replace file="${build.temp}/MANIFEST.MF">
      <replacefilter token=":" value="=" />
      <replacefilter token=":" value="=" />
      <replacetoken>;</replacetoken>
      <replacevalue>
      </replacevalue>
    </replace>
    <property file="${build.temp}/MANIFEST.MF"/>

    <!-- Plugin locations -->
    <property name="plugin.jarname" value=
      "com.qualityeclipse.favorites_${Bundle-Version}" />
    <property name="plugin.jar" location=
      "${build.temp}/jars/plugins/${plugin.jarname}.jar" />
    <property name="product.zip" value=
      "${build.out}/Favorites_v${Bundle-Version}.zip" />

    <!-- Assemble plug-in JAR -->
    <mkdir dir="${build.temp}/jars/plugins" />
    <zip destfile="${plugin.jar}">
      <zipfileset dir="bin" />
      <zipfileset dir="." includes="META-INF/MANIFEST.MF" />
      <zipfileset dir="." includes="plugin.xml" />
      <zipfileset dir="." includes="icons/*.gif" />
      <zipfileset dir="." includes="src/**/*.*" />
    </zip>

    <!-- Assemble the product zip -->
    <zip destfile="${product.zip}">
      <fileset dir="${build.temp}/jars" />
```

```
</zip>
```

```
</target>
```

```
</project>
```

为了执行该Ant脚本，右键点击build-favorites.xml文件并选择Run Ant... (图2-16)。当Ant向导出现时，在JRE选项卡上点击并选择Run in the same JRE as the workspace选项 (图2-17)。点击Run按钮以构建产品。

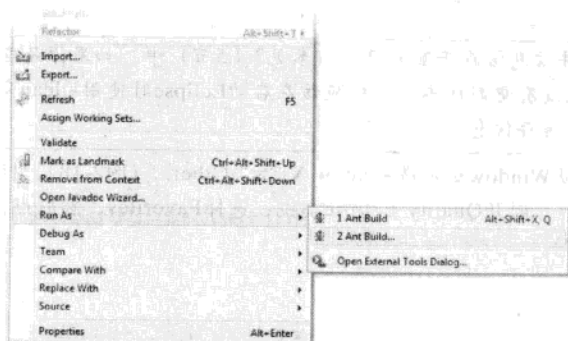


图2-16 build.xml弹出菜单

提示 如果你的Ant脚本使用了Eclipse特定的Ant任务，如`pdex.exportPlugins`，那么你必须选择Run in the same JRE as the workspace选项，这样Ant脚本才能正确执行。

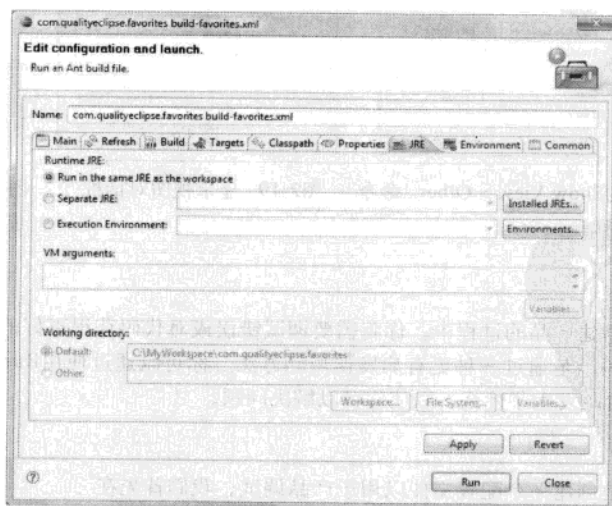


图2-17 Ant向导

2.5 安装并运行产品

为了安装Favorites插件，请执行以下步骤：

- 关闭Eclipse。
- 将Favorites_v1.0.0.zip文件解压至你的Eclipse目录（如C:/eclipse）。
- 确认收藏夹插件存在于/plugins目录中（比如，C:/eclipse/plugins/ com.qualityeclipse.favorites_1.0.0.jar）。
- 重新启动Eclipse。

提示 Eclipse将插件信息缓存于配置目录（参见3.4.5节）中。如果你是在已安装插件上覆盖一个较新的版本，且没有更新版本号，那应该在启动Eclipse时使用-clean命令行选项。这样才能让Eclipse重建插件缓存信息。

当Eclipse重启后，从Window菜单选择Show View > Other...（图2-18），打开Show View对话框（图2-19）。在该对话框中，展开Quality Eclipse类别，选择Favorites，然后点击OK按钮。这样将打开Favorites视图（图2-20）。

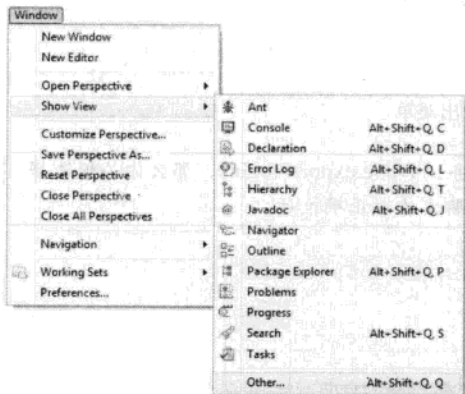


图2-18 Window菜单的Show View > Other...命令

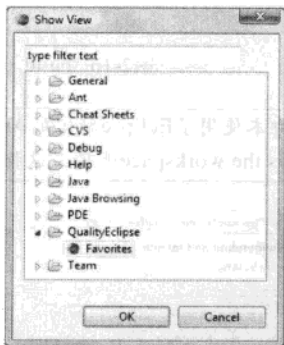


图2-19 显示视图对话框

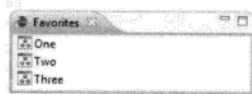


图2-20 收藏夹视图的最初、最简单的样式

2.6 调试产品

不可避免地，在创建产品的过程中，你将需要调试错误或对代码获得更好的了解。这种了解不是通过仅仅查看源代码而是通过一种更有启发作用的方法。你可以通过使用Runtime Workbench来精确查看在产品执行阶段发生了什么，以使你可以解决问题。

2.6.1 创建配置文件

该过程的第一步是创建一个配置文件以用于产品调试。我们首先在Debug工具栏菜单中选择Debug Configurations...（图2-21）。

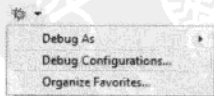


图2-21 调试菜单

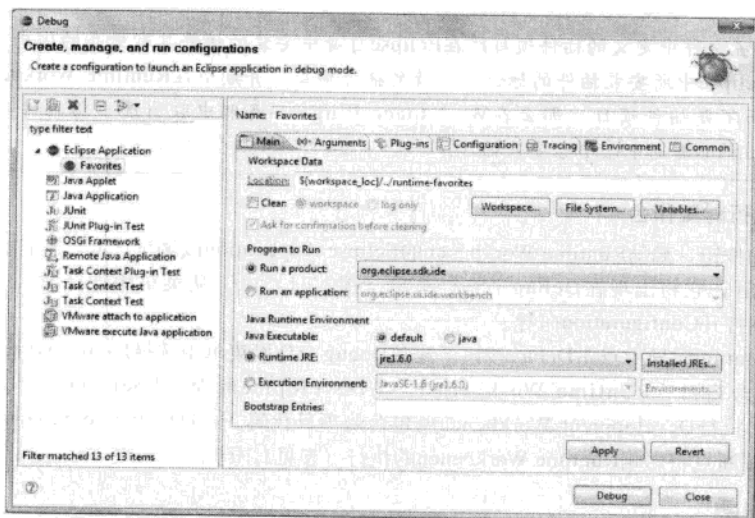



图2-22 定义新的配置文件

在出现的对话框中（图2-22），选择Eclipse Application，然后点击New  按钮。然后，输入“Favorites”作为配置文件的名称。

2.6.2 选择插件和片段

选择Plug-ins选项卡，然后选择Launch with复选框中的plug-ins selected below only（图2-23）。在插件列表中，确认在Workspace Plug-ins类别中选中Favorites插件，而不是在External Plug-ins类别中。

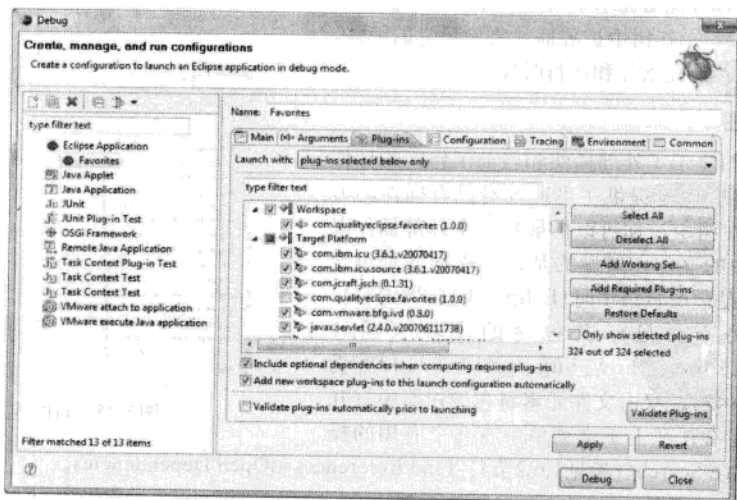


图2-23 在配置文件中选择插件

提示 在配置文件中定义的插件项目比在Eclipse自身中安装的插件具有更高的优先级。如果你有一个与Eclipse中所安装插件的标识符一致的插件项目，并期望在Runtime Workbench使用已安装插件而不是插件项目，那么在Workspace Plug-ins类别中取消选中该插件项目，并在External Plug-ins类别中选中已安装插件。

2.6.3 启动运行时工作台

点击Debug按钮，启动Runtime Workbench的Eclipse Application以调试产品。当你已经定义并使用过配置文件后，它将出现在Debug工具栏菜单（图2-21）中。从菜单中选择它，启动Runtime Workbench而不打开Configuration向导。

在Configuration向导中点击Debug按钮，或从Debug工具栏菜单中选择Favorites后，Eclipse将打开第二个工作台窗口（Runtime Workbench，与Development Workbench相对）。该Runtime Workbench窗口执行Development Workbench所包含的项目代码。在Development Workbench中所做出的更改和设置断点将影响Runtime Workbench的执行（参见1.10节以了解更详细的信息）。

2.7 PDE视图

插件开发环境（Plug-in Development Environment, PDE）提供了用于监视插件不同方面的数个视图。要打开不同的PDE视图，选择Window > Show View > Other...，在Show View对话框中，展开PDE和PDE Runtime两个类别。

2.7.1 插件注册表视图

Plug-in Registry视图显示了当前工作区所有插件的树形视图（图2-24）。在树中扩展插件将显示它的组件，如扩展点、扩展、前提条件和运行时库。

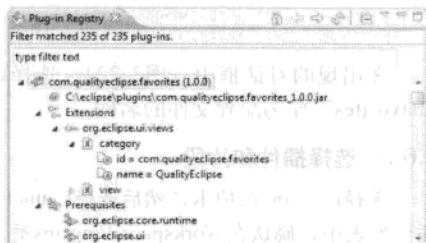


图2-24 插件注册表视图

2.7.2 插件视图

Plug-ins视图显示了外部插件和当前工作区的插件项目的树形列表。它还提供了迅速查看已有插件的方法（图2-25）。在树中，你可以扩展每一个外部插件以浏览插件目录中的文件。遗憾的是，如果插件是包含于JAR文件而不是一个目录中时（Eclipse 3.1新增内容），在该视图中将不会显示其包含的文件（通过bugs.eclipse.org/bugs/show_bug.cgi?id=89143查看Bugzilla条目89143）。双击文件元素将在编辑器中打开该文件以用于查看，在上下文菜单中有几个有用的操作，如Add to Java Search（参见1.6.2节）、Find References和Open Dependencies。

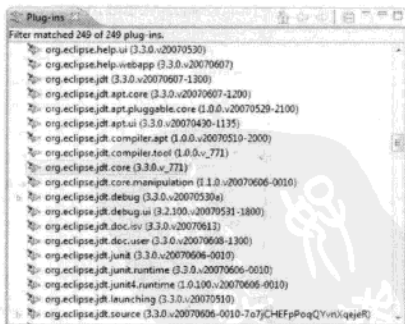


图2-25 插件视图

2.7.3 插件依赖项视图

插件依赖项视图展示了一个关于插件间相互依赖关系的层次视图（图2-26）。当该视图打开后，首先右键点击com.qualityeclipse.favorites插件，然后选择Focus On。双击树中的元素将打开对应的插件清单编辑器。

2.7.4 插件手动搜索

除了上面描述的视图之外，PDE还提供了搜索扩展项引用、扩展点声明和插件的功能。按下Ctrl+Shift+A以打开PDE搜索对话框（图2-27），然后输入插件ID以过滤列表。该对话框还包含扩展项和扩展点的过滤器，以帮助你迅速、简易地查找所需内容。



图2-26 插件依赖项视图

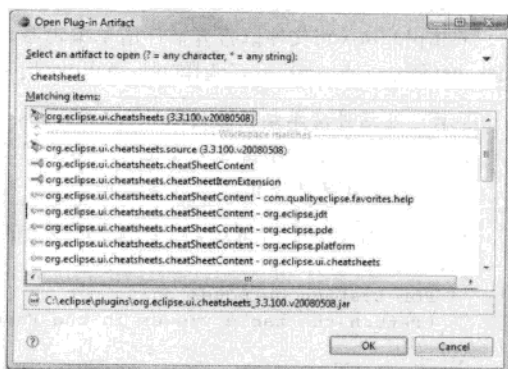


图2-27 PDE插件手动对话框

2.7.5 插件探测器

为了查找当前选中的用户界面元素的更多信息，可以通过Alt+Shift+F1打开插件探测器（Plug-in Spy）（图2-28）。插件探测器（也称为PDE探测器）当前提供了关于选择、编辑器、视图、对话

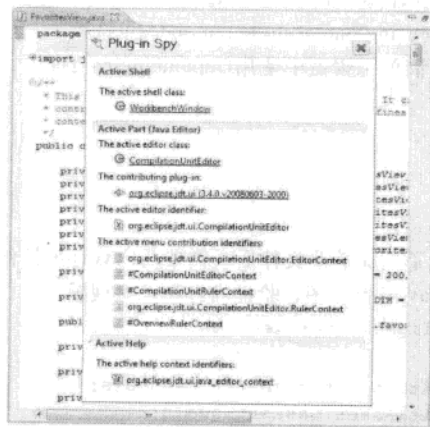


图2-28 插件探测器的弹出窗口

框、首选项页和向导的信息。当查看由插件探测器提供的信息时，点击不同的超链接，将为该插件打开插件清单编辑器。

2.8 编写插件测试

Eclipse在不断地进步。当创建插件时，测试是保证插件在Eclipse不同版本上均能正常工作所必需的。如果目标是一次性开发和发布插件，那么手动测试将会满足需要。然而，自动、手动联合测试对于防止产品随着时间流逝而逐步出现退化有很好的作用。

2.8.1 测试准备

在为Favorites视图创建测试之前，你需要修改收藏夹插件清单，以使合适的类对于测试插件可见。双击plugin.xml文件以打开插件清单编辑器，然后切换至Runtime页面（图2-11）。在Exported Packages部分，点击Add...，选择com.qualityeclipse.favorites.views包，并选择File > Save保存更改。

提示 在插件清单编辑器的Runtime页面，你可以通过在Package Visibility部分设置哪些插件可以访问包来限制导出包的可见性（参见21.2.5节）。或者，你也可以把测试放置在代码片段中，这样就不需要导出包了（想了解更多有关于代码片段的信息，参见16.3节）。

然后，添加恰当的访问方法以使测试可以验证视图内容。在FavoritesView类中，添加以下方法：

```
/**
 * For testing purposes only.
 * @return the table viewer in the Favorites view
 */
public TableViewer getFavoritesViewer() {
    return viewer;
}
```

2.8.2 创建插件测试项目

使用与2.2节中列出的相似过程，来创建一个新的插件项目。以下是与2.2节不同之处：

- 将项目命名为com.qualityeclipse.favorites.test。
- 取消选中Create a plug-in using one of these templates单选框。

在该项目创建后，在插件清单编辑器中使用Dependencies页面（图2-10）以添加如下的所需插件并保存更改：

- com.qualityeclipse.favorites
- org.junit4

2.8.3 创建插件测试

当创建了项目，并且修改了插件清单文件后，需要为Favorites插件创建简单测试（参见以下代码示例）。测试的目标是显示Favorites视图，验证内容和隐藏视图。

```
package com.qualityeclipse.favorites.test;

import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
import org.eclipse.core.runtime.Platform;
```

```
import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.widgets.Display;
import org.eclipse.ui.PlatformUI;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.qualityeclipse.favorites.views.FavoritesView;
/**
 * The class <code>FavoritesViewTest</code> contains tests
 * for the class {@link
 *     com.qualityeclipse.favorites.views.FavoritesView}.
 * @pattern JUnit Test Case
 * @generatedBy CodePro Studio
 */

public class FavoritesViewTest
{
    private static final String VIEW_ID =
        "com.qualityeclipse.favorites.views.FavoritesView";
    /**
     * The object that is being tested.
     *
     * @see com.qualityeclipse.favorites.views.FavoritesView
     */
    private FavoritesView testView;
    /**
     * Perform pre-test initialization.
     */
    @Before
    public void setUp() throws Exception {
        // Initialize the test fixture for each test
        // that is run.
        waitForJobs();
        testView = (FavoritesView)
            PlatformUI
                .getWorkbench()
                .getActiveWorkbenchWindow()
                .getActivePage()
                .showView(VIEW_ID);
        // Delay for 3 seconds so that
        // the Favorites view can be seen.
        waitForJobs();
        delay(3000);
        // Add additional setup code here.
    }
    /**
     * Run the view test.
     */
}
```

```

@Test
public void testView() {
    TableView viewer = testView.getFavoritesViewer();
    Object[] expectedContent =
        new Object[] { "One", "Two", "Three" };
    Object[] expectedLabels =
        new String[] { "One", "Two", "Three" };

    // Assert valid content.
    IStructuredContentProvider contentProvider =
        (IStructuredContentProvider)
            viewer.getContentProvider();
    assertEquals(expectedContent,
        contentProvider.getElements(viewer.getInput()));
    // Assert valid labels.
    ITableLabelProvider labelProvider =
        (ITableLabelProvider) viewer.getLabelProvider();
    for (int i = 0; i < expectedLabels.length; i++)
        assertEquals(expectedLabels[i],
            labelProvider.getColumnText(expectedContent[i], 1));
}

/**
 * Perform post-test cleanup.
 */
@After
public void tearDown() throws Exception {
    // Dispose of test fixture.

    waitForJobs();
    PlatformUI
        .getWorkbench()
        .getActiveWorkbenchWindow()
        .getActivePage()
        .hideView(testView);
    // Add additional teardown code here.
}

/**
 * Process UI input but do not return for the
 * specified time interval.
 *
 * @param waitTimeMillis the number of milliseconds
 */
private void delay(long waitTimeMillis) {
    Display display = Display.getCurrent();

    // If this is the UI thread,
    // then process input.
    if (display != null) {
        long endTimeMillis =
            System.currentTimeMillis() + waitTimeMillis;
    }
}

```

```
while (System.currentTimeMillis() < endTimeMillis)
{
    if (!display.readAndDispatch())
        display.sleep();
}
display.update();
}
// Otherwise, perform a simple sleep.

else {
    try {
        Thread.sleep(waitTimeMillis);
    }
    catch (InterruptedException e) {
        // Ignored.
    }
}
}
}
/**
 * Wait until all background tasks are complete.
 */
public void waitForJobs() {
    while (!Job.getJobManager().isIdle())
        delay(1000);
}
}
```

2.8.4 运行插件测试

创建测试类的后续步骤是设置并执行测试。与创建运行时配置文件类似（参见2.6.1节），创建测试配置文件包括：右键点击Package Explorer的FavoritesViewTest，选择Run As > JUnit Plug-in Test命令。这将自动构建一个测试配置文件并执行测试。然后，你将会看到运行时工作台出现，打开收藏夹视图，运行时工作台关闭。JUnit视图表示你的测试成功执行，Favorites视图内容已被验证（图2-29）。

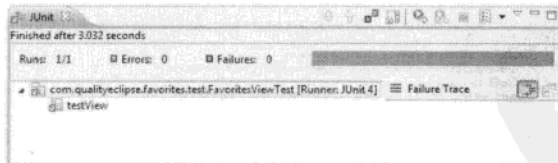


图2-29 JUnit视图

再次右键点击FavoritesViewTest，并选择Run As > Run Configurations...以打开Configuration向导（图2-30）。在这里，你可以设置是单一测试自身执行，还是项目中的所有测试同时执行。Eclipse默认启动产品，这将打开欢迎视图（图1-2）。要更改这一点，点击Main选项卡，并选择Run an application单选按钮。

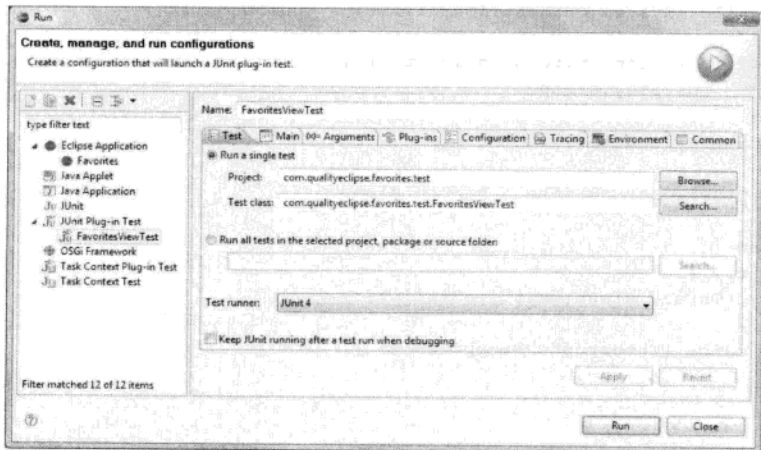


图2-30 测试配置向导

2.8.5 卸载收藏夹插件

可以通过以下步骤来从Development Workspace删除收藏夹插件：

- 1) 关闭Favorites视图。
- 2) 关闭Eclipse。
- 3) 在Eclipse插件目录中删除com.quality.favorites_1.0.0.jar文件。
- 4) 重启Eclipse。如果重启时得到一个错误消息（图2-31），至少有一个Favorites视图没有在第2步中被正确关闭。

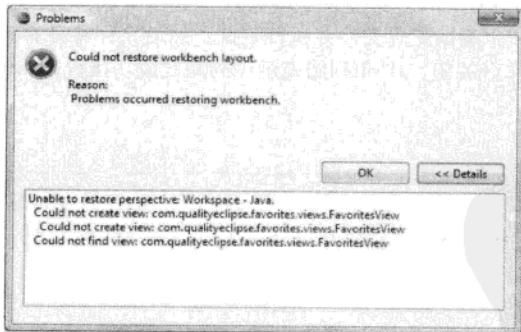


图2-31 Eclipse重启时的错误对话框

- 5) 确认Favorites视图在打开时Show View对话框不再可用（图2-18），并确认Quality Eclipse类别不再存在（图2-19）。

2.9 本书示例

本书中每一章的示例代码都可以下载，并在Eclipse中安装以用于查看。从<http://www.>

qualityeclipse.com下载并安装书中示例，或使用升级管理器（参见18.3.5节）。在升级管理器中输入“<http://www.qualityeclipse.com/update>”，选择Window > QualityEclipse Book Samples以打开视图（图2-32）。

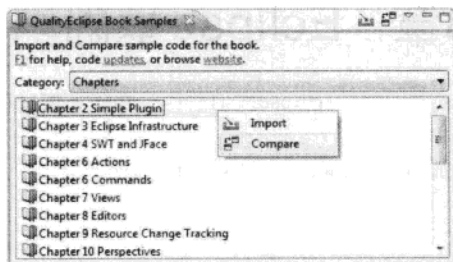


图2-32 QualityEclipse本书示例视图

使用本书示例视图，你可以将工作与示例代码进行比较，并载入特定章的代码以查看。

2.10 总结

这一章包含了创建、运行、调试、监视和测试示例插件的整个过程。后续章节将描述该过程的每一个方面和更多细节。

参考文献

Gamma, Eric, and Kent Beck, *Contributing to Eclipse*. Addison-Wesley, Boston, 2003.

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, Boston, 2005.

FAQ How do I find a particular class from an Eclipse plug-in? (http://wiki.eclipse.org/FAQ_How_do_I_find_a_particular_class_from_an_Eclipse_plug-in%3F).

Helpful tools in PDE Incubator (<http://www.eclipse.org/pde/incubator/>).



第3章 Eclipse基础结构

本章讨论了第2章生成的代码背后的Eclipse基础结构。在深入探讨程序的细节之前，我们先回过头来从整体上对Eclipse进行审视。

在第2章开始编写并描述的简单示例插件——Favorites插件，提供了用来讨论Eclipse基础结构的实例基础。

3.1 结构概述

Eclipse不是一个一整块的程序，而是一个包含插件载入器的、由数百个插件（可能增至数千个）包围的小内核。该小内核是OSGi R4规范的一个实现，为插件执行提供了环境。每一个插件以结构化的方式在整体中发挥作用，可能依赖于由其他插件提供的服务，也可能提供其他插件需要的服务。

这种模块化的设计让Eclipse自身分解为不同的功能块。这些功能块能更容易地被重用于创建超出Eclipse原始开发人员视野之外的应用程序。创建客户端程序的最小插件集称为Eclipse富客户端平台（Eclipse Rich Client Platform，RCP）。当Eclipse基础结构也正被用于创建服务端，Eclipse基础结构也称为Eclipse应用程序框架（Eclipse Application Framework，EAF）。

- EAF——<http://www.eclipse.org/home/categories/frameworks.php>
- OSGi——<http://www.eclipse.org/equinox/>
- RCP——<http://www.eclipse.org/home/categories/rcp.php>

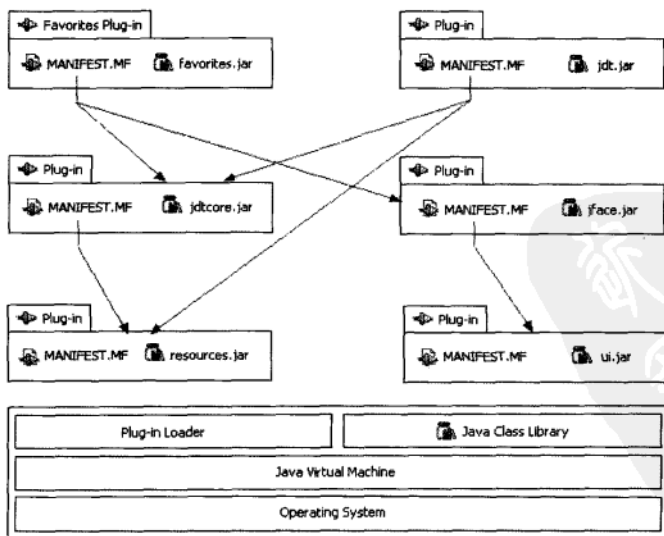


图3-1 Eclipse插件结构和插件间依赖关系示例

3.1.1 插件结构

每个插件的行为都是位于代码中，然而，插件的依赖项和服务（参见2.3.1节）是在MANIFEST.MF和plugin.xml文件中声明的（图3-2）。这种结构从当需要时的角度简化了插件代码的延迟载入，因此也减少了启动时间和Eclipse的内存占用。

在启动时，插件载入器为每一个插件扫描MANIFEST.MF和plugin.xml文件，然后创建一个包含该信息的结构。该结构将占用一定的内存空间，但它允许载入器可以更快地查找所需插件，而且它比一直载入所有插件的所有代码占用的空间要小得多。

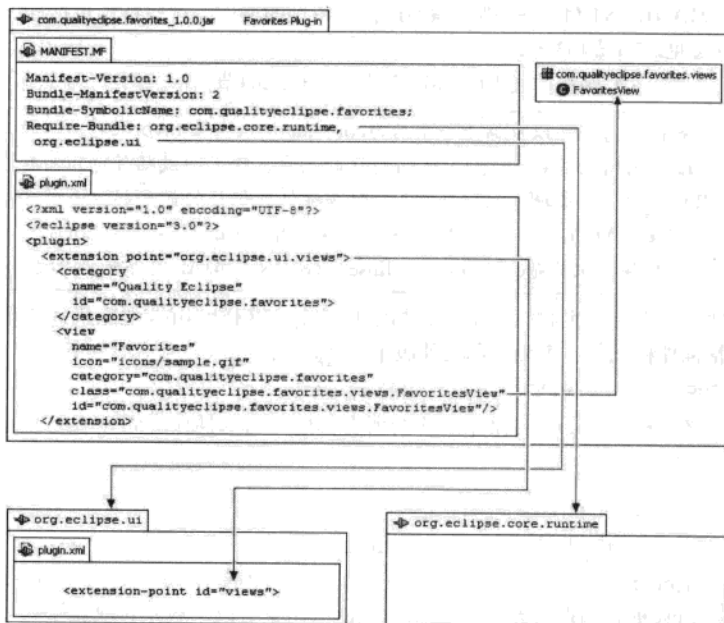


图3-2 声明新的扩展项该示例展示如何在插件清单中声明一个新的扩展，而高亮显示的行展示了插件清单如何引用不同的插件的用户指定项

插件已被载入，但还没有被卸载 在Eclipse 3.1中，插件是在会话中延迟加载并且不被卸载的。这样会造成内存占用随着用户使用更多的功能而增长。在以后版本的Eclipse中，该问题将通过卸载不需要插件而解决（参见eclipse.org/equinox，要查询更多关于使插件无效的规范，参见eclipse.org/equinox/incubator/archive/dynamicPlugins/deactivatingPlugins.html）。

3.1.2 工作区

在Eclipse IDE中可以显示和修改工作区（Workspace）中的文件。工作区是一个目录层次结构。它既包含了用户文件，如项目、源代码等，也包含了插件状态信息，如首选项（参见3.4.4节）。工作区目录层次结构中的插件状态信息只与该工作区相关联。而Eclipse IDE、它的插件、插件静态资源（参见3.4.3节）和插件配置文件（参见3.4.5节）在多个工作区中共享。

3.2 插件目录与JAR文件

Favorites插件JAR, `com.qualityeclipse.favorites_1.0.0.jar`, 包含了与一个典型插件类似的文件, 包括有Java类文件、插件所使用的不同的图像文件和插件清单。

- Java类——组成插件的实际Java类, 位于插件JAR文件的标准Java目录/包结构中。
- 图标 (icon) —— 图像文件一般位于icons或images目录中, 并在plugin.xml中被插件不同的类引用。图像文件和其他静态资源文件作为插件的一部分分发, 可以通过启动器中的方法访问 (参见3.4.3节)。
- META-INF/MANIFEST.MF——描述插件运行时特性的文件。这些特性包括标识符、版本和插件依赖项 (参见2.3.1节和3.3.2节)。
- plugin.xml——描述扩展和扩展点的XML文件 (参见3.3.4节)。

插件目录 收藏夹插件内容可以存储于一个名为`com.qualityeclipse.favorites_1.0.0`的目录中。该目录包含和`com.qualityeclipse.favorites_1.0.0.jar`同样的元素, 除了组成插件的Java类应当存储于插件目录中的JAR文件中。一般地, 该JAR文件应根据插件标识符的最后部分来命名, 但它也可以拥有任何名称, 只要在META-INF/MANIFEST.MF文件中的Bundle-ClassPath声明了名称。在这种情况下, 收藏夹插件标识符是`com.qualityeclipse.favorites`, 那JAR文件应被命名为`favorites.jar`。

插件JAR文件必须有指定名称, 并放置在指定目录中, 这样Eclipse就能找到并载入它。JAR文件的名称必须包括插件标识符、下划线和插件版本, 并以点分隔开:

```
com.qualityeclipse.favorites_1.0.0.jar
```

插件JAR文件必须放在plugins目录中, 作为所有其他Eclipse插件的同等节点, 就如同Favorites插件一样。

3.2.1 链接文件

插件可以通过以下三种方式添加至已有的Eclipse:

- 1) 将插件添加至plugins目录, 并作为其他Eclipse插件的平等节点, 如上一节所示。
- 2) 创建一个Eclipse更新站点 (参见18.3节), 以使Eclipse可以下载并管理插件。
- 3) 将插件放置于一个独立的产品相关的目录, 并创建一个链接文件, 以使Eclipse可以找到并载入这些插件。

第三种方法不仅满足RFRS需求, 还允许多个Eclipse安装版本链接至同一个插件集。你需要对Favorites示例做出几个更改, 这样它才能使用这种方法。

首先, 使用2.8.5节中列出的步骤将已有的Favorites插件从Development Workbench的当前位置移除。然后, 修改基于Ant的build-favorites.xml文件, 这样可以更改Favorites插件为新的结构。这种更改方法为按照以下所述插入QualityEclipse/Favorites/eclipse:

- 1) 替换以下内容:

```
<property name="plugin.jar" location=
    "${build.temp}/jars/plugins/${plugin.dir}.jar" />
```

为 (location必须在一行内):

```
<property name="plugin.jar" location=
    "${build.temp}/jars/QualityEclipse/Favorites/
    eclipse/plugins/${plugin.dir}.jar" />
```

2) 替换以下内容:

```
<mkdir dir="${build.temp}/jars/plugins" />
```

为(所有内容均在一行内):

```
<mkdir dir="${build.temp}/jars/QualityEclipse/Favorites/  
eclipse/plugins" />
```

在进行这些更改时,需要保证location字符串全部在同一行;Ant无法处理有多行的路径。当修改后的build-favorites.xml被执行后,所生成的zip文件包含了新的结构:

```
QualityEclipse/Favorites/eclipse/plugins/  
com.qualityeclipse.favorites_1.0.0.jar
```

zip文件可以被解压缩至任何位置,但对于本示例来说,请保证文件被解压缩至C盘根目录,这样插件的目录是:

```
C:\QualityEclipse\Favorites\eclipse\plugins\  
com.qualityeclipse.favorites_1.0.0.jar
```

Eclipse产品目录位置和Quality-Eclipse产品目录的位置是由用户决定,因此在创建时是未明确的。基于这一点,指向Quality-Eclipse产品目录的链接文件必须在此时手动创建。在Eclipse产品目录(比如,C:\eclipse\links)中创建links子目录,并创建一个文件,命名为com.qualityeclipse.favorites.link。它包含了以下行的内容:

```
path=C:/QualityEclipse/Favorites
```

在Windows中执行这项任务,你可以使用记事本创建并保存文件为一个txt文件。你可以在稍后重命名该文件。请注意,*.link文件中的路径必须使用斜杠(/)而不是反斜杠(\)。新的*.link文件将会在Eclipse重启后被Eclipse使用。

请不要在链接文件中使用相对路径 Eclipse 3.4不允许链接文件包含相对路径。该限制有可能在后续版本中更改(参见bugs.eclipse.org/bugs/show_bug.cgi?id=35037以了解Bugzilla条目35037的信息)。

3.2.2 混合途径

一些产品使用了混合途径,并且可以通过多种形式分发产品。安装时,安装程序将产品插件直接放置于Eclipse插件目录中。然而,当安装至Rational Application Developer或其他Rational IDE产品系列的产品时,产品插件被放置于一个单独的产品目录,并创建了链接文件。此外,这些产品有多种的zip文件格式,每一个分别对应一个特定类型和版本的Eclipse或WebSphere产品。当RFRS认证不需要时,混合路径为Eclipse创建了更简单、更小的zip安装文件。混合路径也为Rational IDE系列产品带来一个更简单的基于安装程序的安装文件。

当你按照上述内容安装了QualityEclipse产品,并创建了链接文件之后,QualityEclipse产品就可以使用了。通过重启Eclipse,打开Favorites视图可以验证是否在新位置正确安装了QualityEclipse产品。当你已经安装并验证了产品之后,请务必通过删除链接文件以卸载它,这样2.8节中描述的JUnit测试仍然可以正常运行。

3.3 插件清单

正如之前所述,每个插件中有两个文件(MANIFEST.MF和plugin.xml)定义了不同的高级内容,这样插件就可以只在你需要其功能才载入。这些文件的格式和内容可以在Eclipse帮助文档中找到。

可以通过打开Help > Help Contents查看Platform Plug-in Developer Guide > Reference > Other Reference Information > OSGi Bundle Manifest和Plug-in Manifest。

OSGi是什么 Eclipse一开始使用了为其量身定做的自身开发的运行时模型/机制。这种机制经过高度优化，并为Eclipse量身定做的，但它不是最佳方案。这是由于还有很多复杂事务需要考虑，并具有一个独特运行时机制阻止了在其他领域重用该机制（如OSGi、Avalon、JMX等）。从Eclipse 3.0开始，引入了新的运行时层。该运行时层是基于从OSGi Alliance而来的技术（www.osgi.org），它具有完善的规范、一个良好的对象模型，支持动态行为，并一定程度上与Eclipse原有运行时类似。随着Eclipse新版本的不断发布，Eclipse运行时API和实现（如“插件”）逐渐向OSGi运行时模型靠拢（如“包”）。

3.3.1 插件声明

在每一个包清单中，包含了名称、标识符、版本、启动器和提供者的条目。

```
Bundle-Name: Favorites Plug-in
Bundle-SymbolicName: com.qualityeclipse.favorites; singleton=true
Bundle-Version: 1.0.0
Bundle-Activator: com.qualityeclipse.favorites.FavoritesActivator
Bundle-Vendor: Quality Eclipse
```

插件清单中的字符串，如插件名，可以被移至单独的plugin.properties文件。该过程简化了国际化的难度（参见第16章）。

1. 插件标识符

插件标识符（Bundle-SymbolicName）被设计用于唯一标识插件，并一般使用Java包命名规则创建的（比如，com.<companyName>.<productName>，在这里是com.qualityeclipse.favorites）。如果几个插件是同一产品的不同部分，那每一个插件名称可包含四五部分，如同com.qualityeclipse.favorites.core和com.qualityeclipse.favorites.ui中的那样。

2. 插件版本

每一个插件使用由句号分隔的三个数字表示其版本号（Bundle-Version）。第一个数字表示了主版本号，第二个表示了次级版本号，第三个表示了服务级别，如1.0.0。你可以指定一个可选择的限定词。该限定词可以包含文字或数字字符，如1.0.0.beta_1或1.0.0.2008-06-26（没有空格）。在启动时，如果有两个插件具有同样的标识符，Eclipse会通过先比较主版本号，再次比较级版本号，然后比较服务级别，最后比较限定词（如果已有的话）的原则选择“最新的”插件。

提示 为了解当前使用版本号的大纲和使用插件版本编号的建议以更好的表达兼容性级别，参见eclipse.org/equinox/documents/plugin-versioning.html和wiki.eclipse.org/index.php/Version_Numbering。

3. 插件名称和提供者

名称和提供者均是可读文本，所以它们可以是任意内容并且不必是唯一的。要查看名称、版本和当前已安装插件的提供者，选择Help > About Eclipse SDK，打开About对话框（图3-3），然后点击Plug-in Details按钮以打开Plug-ins对话框（图3-4）。

提示 在如上所示的About Eclipse SDK Plug-ins对话框中的第一行显示了是否指定了一个特定的插件JAR文件。为了了解更多关于指定JAR的信息，参见java.sun.com/docs/books/tutorial/deployment/jar/signindex.html和java.sun.com/developer/Books/javaprogramming/JAR/sign/signing.html。

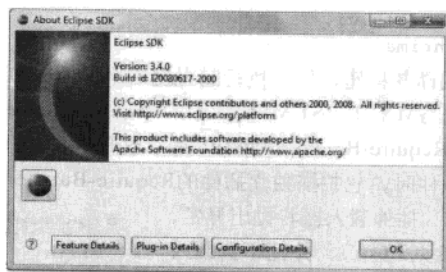


图3-3 关于Eclipse SDK对话框，显示关于Eclipse平台的信息

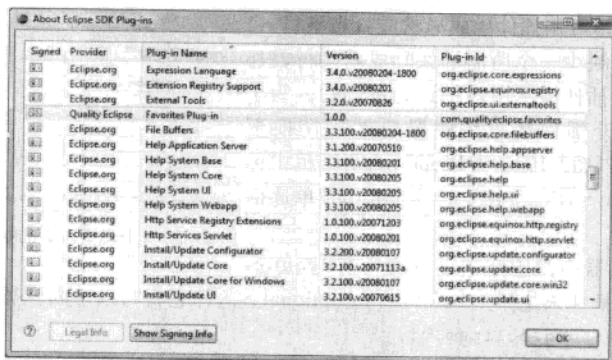


图3-4 关于Eclipse SDK Plug-ins对话框，显示所有已安装插件，并且高亮显示收藏夹插件

4. 插件启动器声明

和Favorites插件类似，每一个插件可以可选地指定一个插件启动器（Bundle-Activator）（参见3.4节）。

3.3.2 插件运行时

MANIFEST.MF文件中的Bundle-ClassPath声明是一个逗号分隔的列表。该列表描述了哪些库（*.jar文件）包含插件代码。Export-Package声明也是一个以逗号分隔的列表。该列表显示了那些库的哪些包可以由其他插件访问（参见21.2.4节和21.2.5节）。

Bundle-ClassPath: favorites.jar

Export-Package: com.qualityeclipse.favorites.views

提示 当将插件如同收藏夹插件那样作为一个单独的JAR文件分发时，Bundle-ClassPath声明应该被省略，这样Eclipse就会在插件JAR中查找类，而不是在插件内部的JAR中查找。

3.3.3 插件依赖项

插件载入器为每一个已载入的插件实例化一个独立的类载入器，并使用清单文件的Require-Bundle声明以决定哪些其他插件（因此也包括那些类）在执行时对于该插件可见（参见21.9节，以了解关于载入未在Require-Bundle声明中指定的载入类的更多信息）。


```
Require-Bundle: org.eclipse.ui,
               org.eclipse.core.runtime
```

如果一个插件已经成功编译并构建，但在执行时抛出一个NoClassDefFoundError异常，这可能表示插件项目的Java classpath与MANIFEST.MF文件中的Require-Bundle声明不一致。正如2.3.1节中所讨论的那样，将classpath和Require-Bundle声明保持同步是十分重要的。

当插件载入器将要载入插件时，它扫描独立插件的Require-Bundle声明并查找所有所需的插件。如果有一个所需插件不可获取，插件载入器将抛出异常，并在日志文件中生成条目（参见3.6节），并且不会载入该独立插件。当一个插件收集扩展它定义的扩展点的所有插件列表时，它不会收集任何不可用的插件。在这种情况下，不会为这些不可用插件生成异常或日志条目。

如果插件可以在缺少一个所需插件时成功执行，那么该所需插件可以在插件清单中被标记为可选的。为了完成这项任务，打开插件清单编辑器，然后切换至Dependencies选项卡（图2-10）。在Properties对话框中选择所需插件，点击Properties按钮，选中Optional单选框（图3-5）。

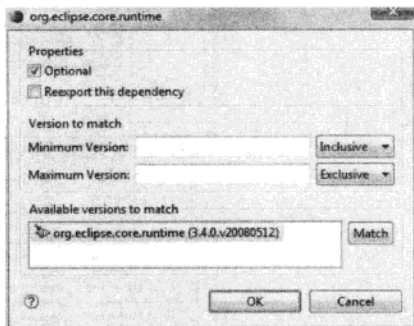


图3-5 所需插件属性对话框

在插件清单编辑器中做出此项更改将为在Require-Bundle声明中的所需插件末尾添加;resolution:=optional。这样它将与以下内容类似：

```
Require-Bundle: org.eclipse.ui,
               org.eclipse.core.runtime;resolution:=optional
```

如果你的插件不仅需要另一个插件的任何版本，在插件清单编辑器的Dependencies选项卡中选择该插件，并点击Properties...按钮。这将打开所需插件的属性对话框（图3-5）。在这里你可以使用Minimum Version和Maximum Version字段来指定一个版本或一段版本区间。更改一个或两个字段，点击OK按钮将对Require-Bundle声明做出更改，如下所示：

```
Require-Bundle: org.eclipse.ui,
               org.eclipse.core.runtime;bundle-version="[3.3.0,3.4.0)"
```

以上内容表示org.eclipse.ui的所有版本和org.eclipse.core.runtime的3.3.x版本都是必需的。另一个示例包括：

- [3.3.0.test,3.3.0.test]——需要一个特定的版本
- [3.3.0,3.3.1)——需要版本3.3.0.x
- [3.3.0,3.4.0)——需要版本3.3.x
- [3.3.0,3.5.0)——需要版本3.3.x或3.4.x
- [3.0.0,4.0.0)——需要版本3.x
- 3.3.0——需要版本3.3.0或更高版本

范围的一般语法为

```
[ floor , ceiling )
```

其中，floor是最小版本号，ceiling是最大版本号。第一个字符可以是[或(，最后一个字符可以是]或)。这些字符表示如下含义：

- [= floor包含在范围之内
- (= floor不包含在范围之内
-] = ceiling包含在范围之内
-) = ceiling不包含在范围之内

你可以指定一个floor或最小版本号而没有其他字符以表示你的插件需要任何大于等于所指定版本号的本号。

最终，选中Reexport this dependency单选框（图3-5）以指定独立的插件类对于该插件的用户是可用的（被（重新）导出的）。默认地，独立类是不被导出的（即它们是不可见的）。

除了Import-Package指定了执行时所需要的包名称而不是bundle的名称之外，Import-Package与Require-Bundle是类似的。使用Import-Package可以视为考虑所需服务的结果，然而使用Require-Bundle与指定服务提供者类似。Import-Package使得为提高同样服务的一个bundle换成另一个变得更容易。但是，这样要知道哪一个提供该服务就变得更困难了。

3.3.4 扩展项与扩展点

插件声明扩展点以使其他插件可以以一种可控的方式扩展原始插件的功能（参见17.1节）。这种机制提供了一个分隔层。在创建原始插件时，原始插件不需要知道扩展插件的存在。插件声明扩展点为它们插件清单的一部分，如同org.eclipse.ui插件中声明的视图扩展点一样：

```
<extension-point
  id="views"
  name="%ExtPoint.views"
  schema="schema/views.exsd"/>
```

你可以在Eclipse帮助文档中找到关于该扩展点的文档（选择Help > Help Contents，在Help对话框中选择Platform Plug-in Developer Guide > Reference > Extension Points Reference > org.eclipse.ui.views）。它表示任何使用该扩展点的插件必须提供实现org.eclipse.ui.IViewPart接口的类的名称（参见21.5节）。

其他插件为原始插件的功能声明扩展项与Favorites插件视图扩展项类似。在这种情况下，Favorites插件使用名称Quality Eclipse和类com.qualityeclipse.favorites.views.FavoritesView声明了视图的一个新类别，作为如下所示的一个新类型的视图：

```
<extension point="org.eclipse.ui.views">
  <category
    name="QualityEclipse"
    id="com.qualityeclipse.favorites">
  </category>
  <view
    name="Favorites"
    icon="icons/sample.gif"
    category="com.qualityeclipse.favorites"
    class="com.qualityeclipse.favorites.views.FavoritesView"
    id="com.qualityeclipse.favorites.views.FavoritesView">
  </view>
</extension>
```

扩展点的每种类型可能需要不同的属性以定义扩展项。一般地，ID属性与插件标识符类似。类别ID为Favorites视图惟一性定义包含其在内的类别提供了方法。类别和视图的name属性都是可读文本，

而icon属性指定了从插件到与视图相关联的图像文件的相对路径。

这种实现允许Eclipse载入不同插件声明的关于扩展项的信息，而不载入插件本身。因此减少了所需操作的时间和内存占用。比如，选择Windows > Show View > Other...打开一个显示所有已知插件提供的全部视图的对话框（参见2.5节）。由于每种类型的视图是在它的插件清单中声明，Eclipse运行时可以不载入包含该视图的所有插件而向用户展示视图列表。

3.4 启动器或插件类

默认地，Bundle-Activator或插件类提供了在插件内部访问静态资源的方法，也提供了访问和初始化插件相关的参数和其他状态信息的方法。启动器不是必需的，但如果在插件清单中指定了启动器，那么该启动器将成为插件载入后第一个被通知的类，和插件准备关闭时最后一个被通知的类（参见3.5.2节和2.3.2节中列出的源代码）。

提示 在过去，插件将它们的启动器暴露为一个条目点。为了更好地控制对插件的初始化和内部资源的访问，请考虑将公共访问方法移至一个新类并隐藏启动器。

3.4.1 启动与关闭

当插件通过start()方法被载入并通过stop()方法关闭时，插件载入器将通知启动器。这些方法允许插件保存和存储Eclipse会话之间的所有状态信息。

当覆盖start()和stop()时请慎重 当覆盖这些方法时，请慎重。总是调用超类实现，并且只采取最少的必需操作，这样你就不会影响Eclipse启动或关闭的速度和内存要求。

3.4.2 插件早期启动

Eclipse惰性地载入插件，因此在启动时，它可能不会调用start()方法。Eclipse可以提供资源更改信息以在插件为不活动时表示所发生的更改（参见9.5节）。如果这还不够并且插件必须在Eclipse启动时载入并启动，插件可以通过向其插件清单中插入以下内容来使用org.eclipse.ui.startup扩展点：

```
<extension point="org.eclipse.ui.startup">
  <startup class="myPackage.myClass" />
</extension>
```

要完成这项工作需要myPackage.myClass类实现org.eclipse.ui.IStartup接口，这样工作台就可以在UI完成启动后立即调用earlyStartup()方法。为了了解更多有关于早期启动和相关信息，参见21.10节。

和大部分插件一样，Favorites插件并不需要在Eclipse启动时载入并运行。因此它不需要使用该扩展点。如果需要早期启动，请在单独插件中只放入必需内容，并且在该单独插件中使用早期启动扩展点，这样附加的早期启动的头部对于启动时间和内容要求只有很小的影响。

3.4.3 静态插件资源

插件可以包含图像和其他基于文件的资源。这些资源和插件清单、库文件一起安装于插件目录。这些文件是静态的，并在多个工作台实例间共享。插件清单中的声明，如操作、视图和编辑器，可以引用资源，如存储于插件安装目录中的图标。起初，插件可以通过使用Plugin类中的启动器方法访问这些资源。这些方法包括find(IPath path)openStream(IPath file)，但这些方法已经不建议使用了。作为替代，根据3.5.2节中描述的那样获取Bundle示例，然后调用org.eclipse.core.runtime.FileLocator

方法，比如：

- `find(Bundle bundle, IPath path, Map override)`——如果指定了包和路径，将返回一个统一资源定位器（Uniform Resource Locator, URL），如果URL无法被解析或创建，则返回null。
- `openStream(Bundle bundle, IPath file, boolean substituteArgs)`——返回指定文件的输入流。文件路径必须是相对于插件安装位置的。如果插件是作为一个单一JAR分发，如收藏夹插件，那么这将是插件JAR内的资源的路径。
- `resolve(URL url)`——解析相对于插件的URL为Java类库的本地URL（如file、http等）。这将替代已不建议使用的Platform方法`resolve(URL url)`。
- `toFileURL(URL url)`——将插件相对URL转换为使用文件协议的URL。这将替代不建议使用的Platform方法`asLocalURL(URL url)`。

3.4.4 插件首选项

插件首选项和其他工作区相关的状态信息都存储于工作区元数据目录层次结构中。比如，如果工作区的位置是C:\eclipse\workspace，那么，Favorites插件参数应存储在：

```
C:/eclipse/workspace/.metadata/.plugins/  
org.eclipse.core.runtime/.settings/  
com.qualityeclipse.favorites.prefs
```

启动器提供了访问插件首选项和其他状态相关文件的方法（参见21.3节以了解更多访问首选项的方法），如下所示：

- `getPreferenceStore()`——返回该插件的首选项（参见3.4.6节以了解关于保存首选项的信息）。
- `getStateLocation()`——返回该插件的插件状态区域位于本地文件系统的位置（参见7.5.2节）。

如果在这次调用之前插件状态区域不存在，将会创建一个插件状态区域。

你可以通过插件目录中的`preferences.ini`文件来为插件提供默认参数（参见12.3.4节）。这种方法可以让你很轻松地通过使用`preferences.properties`文件（参见16.1节）国际化插件。或者，你也可以在程序中提供默认参数，如同12.3.3节描述的那样。

已过时的首选项存储方法 Plugin提供了其他首选项存储方法和类，这种方法是不应使用的：

```
· getPluginPreferences()  
· initializeDefaultPluginPreferences()
```

这些方法存在仅仅是为了向后兼容性。如果使用了它们，将导致额外的Eclipse"legacy"兼容性插件的载入。作为替代，使用AbstractUIPlugin方法，如`getPreferenceStore()`或使用12.3节中列出的`ScopedPreferenceStore`，直接访问首选项。

3.4.5 插件配置文件

如果你需要存储插件信息以在某个特定Eclipse的所有工作区之间共享，那么使用方法`Platform.getConfigurationLocation()`并创建一个插件相关的子目录。如果Eclipse安装于只读位置，那么，`Platform.getConfigurationLocation()`将返回null。你可以将以下字段和方法添加至Favorites Activator类以返回该插件的配置目录。如果Eclipse安装于只读位置，那么这种方法将通过返回工作区相关的状态位置来平滑地降级，而不是返回配置目录，这样插件状态信息仍然可以存储和获取。

```

public File getConfigDir() {
    Location location = Platform.getConfigurationLocation();
    if (location != null) {
        URL configURL = location.getURL();
        if (configURL != null
            && configURL.getProtocol().startsWith("file")) {
            return new File(configURL.getFile(), PLUGIN_ID);
        }
    }
    // If the configuration directory is read-only,
    // then return an alternate location
    // rather than null or throwing an Exception.
    return getStateLocation().toFile();
}

```

首选项也可以通过添加以下字段和方法至FavoritesActivator类来存储于配置目录中。

```

private IEclipsePreferences configPrefs;

public Preferences getConfigPrefs() {
    if (configPrefs == null)
        configPrefs = new ConfigurationScope().getNode(PLUGIN_ID);
    return configPrefs;
}

```

只读安装 要注意如果Eclipse安装于只读位置，那么该方法将会返回null。此外，由以下方法返回的，无论是之后的代码还是参数对象，都不是线程安全的。

如果你将以上方法添加至启动器，那你还应当修改stop()方法，使得在Eclipse关闭时，转储清除配置首选项至磁盘。

```

public void stop(BundleContext context) throws Exception {
    saveConfigPrefs();
    plugin = null;
    super.stop(context);
}

public void saveConfigPrefs() {
    if (configPrefs != null) {
        try {
            configPrefs.flush();
        } catch (BackingStoreException e) {
            e.printStackTrace();
        }
    }
}

```

当你启动运行时工作台时（参见2.6节），你可以通过使用Run对话框的Configuration页指定配置目录（图3-6）。



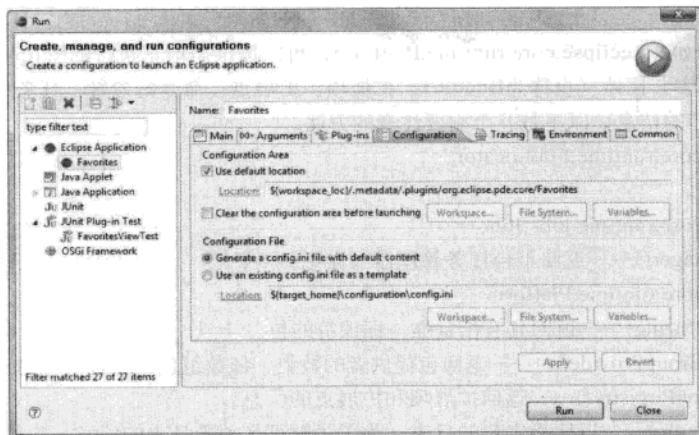


图3-6 用于指定配置目录的启动配置页

3.4.6 插件与AbstractUIPlugin

所有启动器必须实现BundleActivator接口。一般地，基于UI的插件（需要org.eclipse.ui插件的插件）具有一个启动器，该启动器继承AbstractUIPlugin，而非UI插件继承Plugin。两种类都为插件编程人员提供了基本插件服务，但它们之间重要的区别。

当插件关闭时，AbstractUIPlugin自动保存所有插件首选项，这些首选项可以通过getPreferenceStore()方法访问。当直接继承Plugin类时，需要修改stop()方法以保存你的首选项，以使首选项可以在会话间保存。由AbstractUIPlugin提供的其他方法包括：

- createImageRegistry()——返回该插件的新图像注册表。你可以使用该注册表管理插件经常使用的图像。该方法的默认实现创建了一个空注册表。如果需要，则子类可以覆盖该方法。
- getDialogSettings()——返回UI插件的对话框设置（参见11.2.7节）。对话框设置保存工作台环境中该插件的不同向导和对话框的持久状态数据。
- getImageRegistry()——返回该UI插件的图像注册表（参见4.4.3节和7.7节）。
- initializeImageRegistry(ImageRegistry reg)——使用由插件经常使用的图像来初始化一个图像注册表。
- loadDialogSettings()——为该插件载入对话框设置。这些设置首先从插件元数据目录中的dialog_settings.xml文件查找，然后在插件目录中查找具有相同名称的文件。如果这两项都失败了，将创建一个空设置对象。该方法可以被覆盖，即使在一般情况下是不必要的。

3.5 插件模型

当Eclipse第一次启动时，它扫描所有插件目录，并为它找到的每一个插件建立一个内部模型。这项工作扫描每一个插件清单，而不是载入插件时进行。如果你想要显示关于插件或执行基于特定插件属性的操作，而不想要花费时间和内存来载入插件，那么以下两小节中的方法是十分有用的。

3.5.1 平台

有几个类，如`org.eclipse.core.runtime.Platform`，可以提供当前正执行的Eclipse环境的信息。你可以获取关于已安装插件（也称为Bundle）、扩展项、扩展点、命令行参数、任务管理器（参见21.8节）、安装位置等的信息。以下是几个需要注意的方法。

`org.eclipse.core.runtime.FileLocator`

参见3.4.3节。

`org.eclipse.core.runtime.jobs.Job`

- `getJobManager()`——返回平台任务管理器（参见21.8节）。

`org.eclipse.core.runtime.Platform`

- `getBundle(String)`——返回具有指定唯一标识符的包。
- `getBundleGroupProviders()`——返回包提供者的数组，该数组包含了含有当前已安装包的包组。
- `getExtensionRegistry()`——返回扩展项和扩展点的信息。
- `getLog(Bundle)`——返回指定包的日志。为了了解更多信息关于日志的信息，参见3.6节。
- `getProduct()`——返回Eclipse产品信息。
- `inDebugMode()`——如果Eclipse是运行于调试模式，返回`true`，当用户指定了`-debug`命令行参数时，也返回`true`。

`org.eclipse.core.runtime.SafeRunner`

- `run(ISafeRunnable)`——以保护模式运行给定`runnable`。在`runnable`中抛出的异常将被日志记录并传递至`runnable`的异常处理器。

3.5.2 插件与包

有关于当前已安装插件（也被称为包）的信息，可以通过使用`Platform.getBundleGroupProviders()`或`Platform.getBundle(String)`获取。访问启动器或插件类需要包含要载入的插件，而与Bundle接口交互时不需要包含。如果你已经有一个启动器，如Favorites插件，那么你可以使用如下所示的方法为该插件获取Bundle接口：

```
FavoritesActivator.getDefault().getBundle()
```

当你获取了Bundle对象之后，有几个方法是值得关注的。

- `getBundleId()`——返回包的唯一标识符（long类型），由Eclipse在安装包时分配。
- `getEntry(String)`——返回一个URL，其内容为由指定的“/”分隔的包相对路径名。如果是`getEntry("/")`，则返回包的根目录。这可以提供由通常是只读插件提供的资源的访问方法。相对插件信息应当写入由`Plugin.getStateLocation()`提供的位置。
- `getHeaders()`——返回包的MANIFEST.MF文件定义的头和值的字典（参见3.3.1节）。
- `getState()`——返回插件的当前状态，如`Bundle.UNINSTALLED`、`Bundle.INSTALLED`、`Bundle.RESOLVED`、`Bundle.STARTING`、`Bundle.STOPPING`、`Bundle.ACTIVE`。
- `getSymbolicName()`——返回唯一插件标识符（`java.lang.String`类型），与MANIFEST.MF中的`Bundle-SymbolicName`声明的一致。

插件版本号可以使用`getHeaders()`方法获取，如下所示：

```
public Version getVersion() {
    return new Version((String) getBundle().getHeaders().get(
```

```
org.osgi.framework.Constants.BUNDLE_VERSION));  
}
```

3.5.3 插件扩展项注册表

你可以通过Platform.getExtensionRegistry()方法访问插件扩展项注册表。它包含了插件描述符，每个描述符代表一个插件。注册表提供了以下方法以从不同插件获取信息且不载入它们（参见17.1节以了解创建扩展点的信息）。

- getConfigurationElementsFor(String extensionPointId)——返回所有配置至已标识的扩展点的扩展的所有配置元素。
- getExtensionPoint(String extensionPointId)——返回由在该插件注册表中给定扩展点标识符的扩展点。

以前，扩展项和扩展点在执行时不发生改变，但这一事实正在慢慢地改变，这是由于Eclipse插件模型不断使其自身向OSGi靠拢。如果你对于执行时更改感兴趣，则使用addRegistryChangeListener (IRegistryChangeListener)。

提示 为了了解更多关于插件注册表、启动器和生命周期的信息，请在www.eclipse.org/equinox查看Equinox项目。

3.6 日志

RFRS需求表示异常和其他服务相关信息应该添加至日志文件的末尾。为了减轻这一工作，启动器提供了访问插件日志机制的方法。这种方法通过getLog()方法访问。为了方便，FavoritesLog使用几个功能方法包装了由getLog()方法返回的ILog接口：

```
package com.qualityeclipse.favorites;  
  
import org.eclipse.core.runtime.IStatus;  
import org.eclipse.core.runtime.Status;  
  
public class FavoritesLog {  
    然后的第一组方法是为了方便，将信息、错误消息和异常附加至收藏夹插件日志的末尾。  
    public static void logInfo(String message) {  
        log(IStatus.INFO, IStatus.OK, message, null);  
    }  
    public static void logError(Throwable exception) {  
        logError("Unexpected Exception", exception);  
    }  
    public static void logError(String message, Throwable exception) {  
        log(IStatus.ERROR, IStatus.OK, message, exception);  
    }  
}
```

上面的每一个方法最终将调用下面的方法创建一个状态对象（参见3.6.1节），并将该状态对象附加至日志末尾。

```
public static void log(int severity, int code, String message,  
    Throwable exception) {  
    log(createStatus(severity, code, message, exception));  
}
```



```
}  
public static IStatus createStatus(int severity, int code,  
    String message, Throwable exception) {  
    return new Status(severity, FavoritesActivator.PLUGIN_ID, code,  
        message, exception);  
}  
public static void log(IStatus status) {  
    FavoritesActivator.getDefault().getLog().log(status);  
}
```

log()和createStatus()方法使用以下参数。

- severity——严重性，以下值的其中之一：
IStatus.OK、IStatus.WARNING、IStatus.ERROR、IStatus.INFO或IStatus.CANCEL
- code——插件相关的状态码或IStatus.OK
- message——可读文本，根据当前位置已本地化
- exception——低级别的异常，如果不适合则值为null

提示 为了了解更多关于Eclipse 3.4中的日志的发展情况，参见https://bugs.eclipse.org/bugs/show_bug.cgi?id=147824

3.6.1 状态对象

org.eclipse.core.runtime包中的IStatus类型层次结构提供了用于封装、前推和记录操作结果（如果有异常，异常也将包含在内）的机制。单一错误使用一个Status的实例来表示（参见上面源代码中的createStatus方法），而包含零个或多个的子状态对象的MultiStatus对象代表多个错误。

当创建一个将被多个其他插件使用的框架插件时，根据IResourceStatus和ResourceStatus创建类似的状态子类型是十分有用的。然而，对于Favorites插件，以下的已有状态类型将完成同样任务：

- IStatus——代表操作输出的状态对象。所有CoreExceptions携带状态对象以表示出错位置。状态对象也可以由用于提供失败细节的方法返回（如验证方法）。
- IJavaModelStatus——代表Java模型操作的输出。状态对象用于JavaModelException对象内部以表示出错信息。
- IResourceStatus——代表Resources插件中的资源相关状态，并定义相对状态码常量。由Resources插件创建的状态对象携带它的唯一标识符（ResourcesPlugin.PI_RESOURCES）和这些状态码的其中之一。
- MultiStatus——固定多状态实现，适于实例化或继承。
- OperationStatus——描述请求的状态，该请求将用于执行，撤销或重新执行一个操作（参见6.1节）。
- Status——固定状态实现，适于实例化或继承。
- TeamStatus——从一些组操作返回或作为一些TeamException类型的异常的载荷（payload）。

3.6.2 错误日志视图

Eclipse提供了Error Log视图以用于监视Eclipse日志文件。要打开错误日志视图，选择Window > Show View > Other...，并在Show View对话框中，展开General类别以找到Error Log视图（图3-7）。在一个条目上双击将打开一个显示错误日志条目细节的对话框。如果Eclipse安装于C:\Eclipse，且工作区位于直接的子目录中，那么你可以在C:\Eclipse\workspace\metadata\log找到Eclipse日志文件。

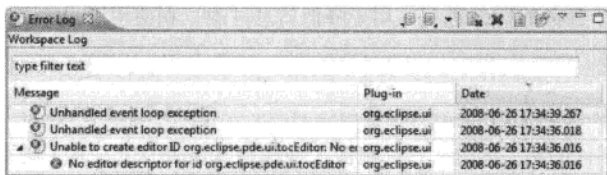


图3-7 由Eclipse平台提供的错误日志视图，显示了当Eclipse运行时生成的信息和异常

3.6.3 处理错误（与其他状态）

org.eclipse.debug.core插件将错误生成从错误处理中分离出来，因此允许非UI插件生成的错误（和其他IStatus对象）被UI插件可选择地处理，不需要强制非UI插件依赖UI插件。任意（一般是UI）插件可以使用org.eclipse.debug.core.statusHandlers扩展点注册一个org.eclipse.debug.core.IStatusHandler。当另一个（一般是非UI）插件需要处理错误或其他IStatus对象，该插件将会调用org.eclipse.debug.core.DebugPlugin.getStatusHandler(IStatus)方法以获取合适的IStatus-Handler。虽然这种状态处理框架特定于org.eclipse.debug.core（不是org.eclipse.core.runtime）插件，但它可以用于（或获取）分离插件。

提示 记录和处理错误的最佳做法不断在进化。为了了解更多信息，参见https://bugs.eclipse.org/bugs/show_bug.cgi?id=200090

3.7 Eclipse插件

插件是基于一个或多个作为Eclipse一部分的基础插件创建的。它们被分散成几个组，并更进一步的分离成UI和核心（Core），如下所示。UI插件包含用户界面的内容或依赖于其他完成该任务的插件，而你可以在无头部的环境（没有用户界面的环境）中使用核心插件。

- Core——普通低级别非UI插件组，这些非UI插件组成了基本服务，如扩展处理（参见第9章）、资源跟踪（参见第17章）等。
- SWT——标准窗口小部件工具集，与底层操作系统紧密联系的UI窗口小部件的通用库，但它具有一个与OS无关的API（参见第4章）。
- JFace——基于SWT创建的附加UI功能的通用库（参见第5章）。
- GEF——Graphical Editing Framework，图形编辑框架减轻了富图形编辑器的开发流程。
- Workbench core——提供Eclipse IDE本身相关的非UI行为，如项目、项目性质和构建器（参见第14章）。
- Workbench UI——提供Eclipse自身相关的UI行为的插件，如编辑器、视图、透视图、操作和首选项（参见第6、7、8、10、12章）。
- Team——提供服务的插件组。这些服务用于集成不同类型的源代码控制管理系统（如CVS）至Eclipse IDE。
- Help——提供作为Eclipse IDE一部分的Eclipse IDE文档的插件（参见第15章）。
- JDT core——Eclipse IDE中的非基于UI的Java开发工具插件。
- JDT UI——Eclipse IDE中的JDT UI插件。
- PDE——插件开发环境（Plug-in Development Environment）。

- Mylyn——专注于任务的UI改进项，用于降低信息过载，和使用用户定义的任务关联信息与该信息的表示。

3.8 总结

本章试图从创建插件的相关内容，让你更深入地理解Eclipse及其结构。随后的两章探讨将用于创建你自己的插件的用户界面元素。

参考文献

“Eclipse Platform Technical Overview,” Object Technology International, Inc., February 2003, (www.eclipse.org/whitepapers/eclipse-overview.pdf).

Melhem, Wassim, et al., “PDE Does Plug-ins,” IBM, September 8, 2003 (www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html).

Xenos, Stefan, “Inside the Workbench: A Guide to the Workbench Internals,” IBM, October 20, 2005 (www.eclipse.org/articles/Article-UI-Workbench/workbench.html).

Bolour, Azad, “Notes on the Eclipse Plug-in Architecture,” Bolour Computing, July 3, 2003 (www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html).

Rufer, Russ, “Sample Code for Testing a Plug-in into Existence,” Yahoo Groups Message 1571, Silicon Valley Patterns Group (groups.yahoo.com/group/siliconvalleypatterns/message/1571).

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.

Buschmann, Frank, et al., *Pattern-Oriented Software Architecture*. John Wiley & Sons, Hoboken, NJ, 1996.

Estberg, Don, “How the Minimum Set of Platform Plug-ins Are Related,” Eclipse Wiki (eclipsewiki.editme.com/MinimumSetOfPlatformPlugins).

Watson, Thomas, “Deprecation of Version-Match Attribute,” equinox-dev email, April 30, 2004.

“Interval Notation” <http://id.mind.net/~zona/mmts/miscellaneousMath/intervalNotation/intervalNotation.html>.

“Interval Mathematics” Wikipedia [http://en.wikipedia.org/wiki/Interval_\(mathematics\)](http://en.wikipedia.org/wiki/Interval_(mathematics)).



第4章 标准窗口小部件工具集

标准窗口小部件工具集 (Standard Widget Toolkit, SWT) 是基于系统原生控件的。SWT为整个Eclipse用户界面提供了基础。本章首先介绍SWT的一些历史和基本原理, 然后深入讨论使用SWT创建应用程序。这部分包括了绝大部分常见的窗口小部件和在窗口中铺放它们的布局管理器。本章最后讨论了在使用SWT时遇到的资源管理事项。

4.1 SWT历史与目标

SWT的根源可以追溯至15年之前, 由对象技术国际组织 (Object Technology International, OTI, 后来成为一家独立的面向对象的先锋性的软件公司, 现在是IBM的一部分) 为Smalltalk完成的 (一开始是为OTI Smalltalk, OTI Smalltalk于1993年成为IBM Smalltalk) 一系列工作, 包括创建多平台、可移植性和原生窗口小部件接口。IBM Smalltalk的通用窗口小部件 (Common Widget, CW) 层提供了快速、原生地访问多个平台的窗口小部件集的方法。它同时也在提供了通用API, 且不用面对“最小公分母 (lowest common denominator, LCD)”问题。而这一问题是在其他可移植的图形用户界面工具集经常需要面对的。

多年以来, IBM已经使用Smalltalk作为它的“秘密武器”来创建开发工具 (IBM的第一个Java IDE, VisualAge for Java, 也是用Smalltalk编写的), 然而Smalltalk的部署和配置问题最终结束了它在IBM的长时间使用。

Java的完全可移植性承诺和无处不在的虚拟机对于负责创建下一代开发工具的IBM工作人员是非常具有吸引力的。OTI在Java身上再一次看到了它可以发挥其特长的另一种语言。

Sun在提供可移植的窗口小部件API, 抽象窗口工具集 (Abstract Windowing Toolkit, AWT) 所做出的初次尝试中, 在两个方面遇到了问题: 对于原生窗口小部件的过于复杂的接口和LCD问题。它提供了对较少的窗口小部件集的访问方法, 包括按钮、标签和列表。大部分平台间都是通用的。但它没有提供对更丰富的窗口小部件, 如表、树和样式文本。基于这种原因, 加上作为一个没有活力的API, 注定了它在市场中的失败。

为了解决AWT的问题并给Java提供一个更强大、可扩展的GUI库, Sun决定放弃原生窗口小部件接口, 并开发它自己的可移植的、仿真的窗口小部件库, 官方名称为Java基础类 (Java Foundation Classes, JFC), 更普遍的名称是Swing。有趣的是, 它与Smalltalk许多年之前的开发是同步的。当时, ParcPlace向世界发布了第一个真正可移植的、多平台的GUI环境。该产品称为VisualWorks (许多负责VisualWorks的可移植、仿真的GUI库的前ParcPlace工程师最终去了Sun公司)。

当Swing通过提供丰富的窗口小部件集解决了LCD问题后, 但平台窗口小部件的模拟还有很多工作要做。Swing应用程序最终成为像Swing应用程序, 但不是平台原生的应用程序。这些应用程序需要被替换。Swing应用程序也在性能上存在一定问题, 因为它们达不到原生的对应部件现有的性能。

AWT可以在Java 2平台, 移动版本 (Java 2 Platform, Micro Edition, J2ME) 设备上运行, 而Swing不行。这是因为庞大的运行时Java虚拟机 (Java Virtual Machine, JVM) 的内存占用, 和

Swing依赖快速原生图像技术来绘制每一个模拟的控件。OTI被赋予在IBM内部为J2ME开发工具的任务。他们认为AWT不是一个足够好的工具集。这是因为AWT只提供了基本控件集，以及它的必须使用JavaBeans组件模型的结构。这种结构允许空的构造。它具有一个占用大量JVM内存的两层对象层。而JVM内存存在小型设备上必须进行很精确的管理。

由于OTI对于Swing和模拟窗口小部件库的基本原理觉得还不够好，而且他们具有关于如何正确创建本地可移植的、多平台的窗口小部件库的丰富经验，OTI开始着手更正AWT和Swing已有的缺陷，并创建一个本来应该由AWT扮演的GUI库。这一工作的结构就是标准窗口小部件集（SWT）。OTI让为Smalltalk开发CW的那些开发人员来为Java开发SWT。

SWT被设计为占有尽可能小的JVM内存。CW由两层组成，包括了一个操作系统层。然而，对于SWT来说，单层结构被认为是更好的选择。每个平台的实现应作为完全优化过的Java类的子集。这些Java类尽可能地直接与原生部件交互。公共API是一样的，但它没有中间层。

OTI使用SWT来开发他们的第一个用Java编写的IDE，VisualAge Micro Edition（VAME）。当IBM决定建立一个通用工具平台（Eclipse）以作为他们那些成功的已有的产品的基础时，他们一开始使用Swing创建。当时所使用的Swing是Java 1.2中早期版本的Swing，IBM对于它的性能和外观感受十分失望。Swing的内存泄漏以及其他缺点，导致IBM最终放弃了它。

SWT被选中的原因之一是由于IBM与Microsoft的正面交锋，而且SWT将带来一个足够丰富的UI体验。这在当时是一个极大的冒险。SWT还没有被移植至很多平台，要接受SWT，用户可能会问：“如果Swing对于你的工具集都不够好，为什么我们要用它？”此外，任何开发插件的人将不得不使用SWT代替Swing。他们的担心在于学习新的API过程中已有的抵触，以及SWT与Swing的对抗将分裂Java社区。这些担心最终都成为了现实。

然而，SWT发现它在Eclipse富客户端平台（Rich Client Platform，RCP）中开发应用程序的人们中获得了大量的欢迎，这是由于他们喜欢它更快的速度和平台集成。Sun在Swing 1.2和1.3版本中可能没有正确地判断当时的形势。在JDK 1.4至1.6中，Sun的Swing的性能和它的外观感受类得到了大幅度的提高。因此，现在使用它的开发者们拥有了一个得到很好改进的工具集。

即使没有SWT成为新标准的威胁，要搞清楚Sun是否已完成这项工作以迎头赶上也是十分困难的。对于两者的用户来说，同时拥有这两种工具集是比较好的选择。在过去，这两个工具集之间的互操作性是不够好的，即使这从Eclipse 3.0开始得到了大幅度的改进。

SWT是整个Eclipse UI的基础。它是快速的、原生的和多平台的，但它没有AWT的LCD问题和Swing的外观感受问题。SWT通过一个两全其美的方法完成了这项工作：它在一个平台上尽可能的使用原生窗口小部件，并在没有相应窗口小部件的平台上模拟以补全它们。一个很好的这方面的示例是树形窗口小部件，它在Windows中是原生，但在Linux中是通过模拟实现的。结果就是一个丰富的、可移植的API，并可以用来创建和所支持的每一个平台的外观感受联系十分紧密的GUI应用程序。

注意 在提供一个一致、高级别、公开的API的情况下，SWT在每个平台的底层实现是区别很大的。SWT对于每一个平台有一个独特的实现，并且低级别的SWT API与它们平台的对应部件是一一对应的关系。为了了解关于SWT如何与本地平台交互的细节讨论的信息，参见www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html。

4.2 SWT窗口小部件

SWT提供了一个丰富的窗口小部件集，可以用于创建独立运行的Java应用程序或Eclipse插件。在了解你可能用到的每一个窗口小部件的细节之前，看看一个简单的独立SWT示例是十分具有指导意义的。

4.2.1 简单独立示例

让我们重新查看在第1章中创建的简单Java项目和HelloWorld应用程序开始。

1. 将SWT添加至你项目的classpath

在你开始使用SWT之前，SWT库需要被添加至你项目的classpath。要添加SWT支持，请完成以下步骤：

- 1) 右键点击该项目，选择Properties命令以打开Properties对话框。
- 2) 选择Java Build Path > Libraries选项卡，并点击Add External JARs按钮。
- 3) 选择至你的Eclipse/plugins目录。
- 4) 选择org.eclipse.swt.win32.win32.x86_3.n.n.vnnnn.jar（或对应的Linux或OS X JAR文件），点击OK按钮以完成添加SWT库至项目的classpath（图4-1）。

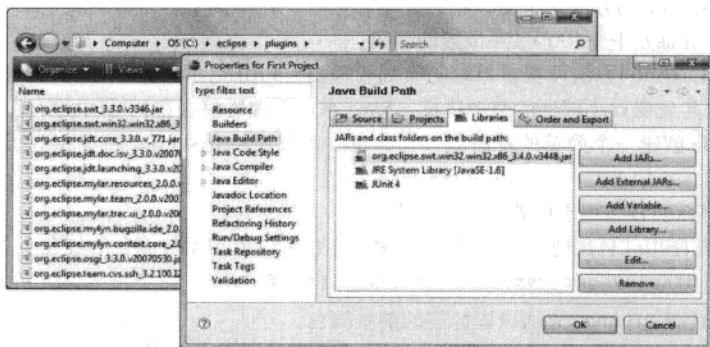


图4-1 Java Build Path > Libraries properties

2. 独立SWT代码

然后，修改HelloWorld类以将其转换为一个独立SWT示例。为了完成这项工作，需要移除main()方法的内容，并用以下代码替代：

```
1 public static void main(String[] args) {
2     Display display = new Display();
3     Shell shell = new Shell(display);
4     shell.setText("Hello World");
5     shell.setBounds(100, 100, 200, 50);
6     shell.setLayout(new FillLayout());
7     Label label = new Label(shell, SWT.CENTER);
8     label.setText("Hello World");
9     Color red = new Color(display, 255, 0, 0);
10    label.setForeground(red);
}
```

```
11     shell.open();
12     while (!shell.isDisposed()) {
13         if (!display.readAndDispatch()) display.sleep();
14     }
15     red.dispose();
16     display.dispose();
17 }
```

注意 在输入以上新的方法内容后，请选择Source > Organize Imports命令（或按下Ctrl+Shift+O）为所有被引用的SWT类添加导入。

以下对每一行进行了详细说明。

第2行——每一个基于SWT的应用程序只有一个Display实例。该实例表示了底层平台和SWT之间的连接。除了可以管理SWT事件循环之外，它还提供访问SWT所需的系统资源的方法。它将在第16行被释放。

第3行——每个窗口有一个Shell对象，它代表用户交互的窗口帧。它处理对所有窗口都类似的移动和变换大小行为，并且它是显示在它边框内的所有窗口小部件的父对象。

第4行——setText()方法用于设置窗口帧的标题。

第5行——setBounds()方法用于窗口帧的大小和位置。在本例中，窗口帧将是200像素宽，50像素高，并位于距离屏幕左上角100×100像素的位置。

第6行——setLayout()方法设置窗口帧的布局管理器。FillLayout是一个简单的布局，它让单一子窗口小部件填充其父窗口小部件的整个边框。SWT布局管理器将会在4.3节中进行深入讨论。

第7行——本行创建一个简单的标签部件，该部件的父部件是shell，并在相对于其自身的中央位置显示它的文本。

第8行——setText()方法用于设置标签的文本。

第9行——本行使用红色创建了一个Color实例。你也可以使用系统颜色的红色。

```
Color red = display.getSystemColor(SWT.COLOR_RED);
```

第10行——setForeground()方法设置标签的前景颜色。

第11行——到此刻为止，窗口帧还不是可见的。open()方法让窗口帧显示。

第12行——while循环不断检查窗口帧是否被关闭。

第13行——display管理事件循环。readAndDispatch()方法从系统事件队列读取事件，并将它们分发至适合的接收者。当还有更多任务要完成时，方法返回true，当事件队列为空时，返回false（此时可以运行UI线程睡眠，直至有更多的任务要完成）。

第15、16行——当循环检测到窗口已经被释放时，就需要释放颜色、display和所有相关联的系统资源。请注意系统颜色不应被释放。

3. 运行示例

要启动Java应用程序，使用Run As > Java Application命令。这样将会创建一个Java应用程序启动配置文件（Java Application launch configuration）（图4-2），该文件可以在Run对话框中被选中。

点击对话框的Run按钮以启动该Java程序（图4-3）。

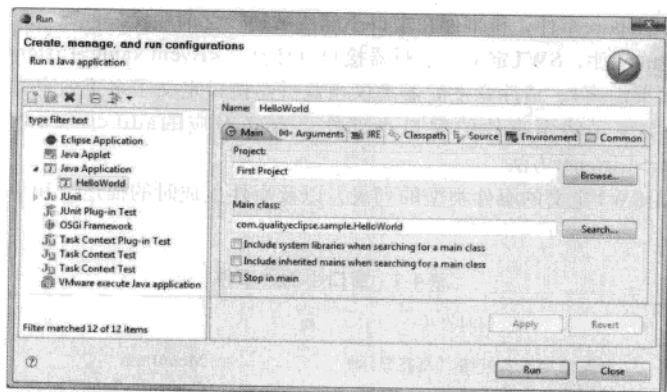


图4-2 运行对话框

4.2.2 窗口小部件生命周期

SWT的目标之一是小而精。要达到这一目标，一个基本设计原则是尽可能多地将窗口小部件状态存储于系统窗口小部件而不是SWT窗口小部件。这一点与Swing是明显不同的。Swing将整个窗口小部件状态维持于窗口小部件内部。通过不复制在系统级维持的信息，SWT窗口小部件是很小的，并且只具有适中的内存占用。

这种实现的一个代价是，SWT窗口小部件不能独立存在。当一个SWT被创建时，它对应的底层系统窗口小部件将立即被创建。几乎所有窗口小部件状态信息请求都传递至系统窗口小部件。

绝大部分系统要求窗口小部件必须在一个特定的父部件中创建，因此SWT要求提供一个父部件作为它的构造函数的参数之一。许多系统的另一个要求是在创建时必须提供一定的样式设置（比如，按钮可以是单选框、单选按钮或简单按钮，而文本框可以是一行的或多行的）。

样式块（Style bit）由SWT类中定义的int常量表示。然后，对样式进行或操作，作为另一个构造函数参数传递以创建窗口小部件的初始样式。请注意不是所有的样式在所有的系统上都被支持。因此，在很多时候，被请求的样式是作为建议，而不一定在某个特定系统中发挥作用。

另一个对SWT产生影响的系统要求是，系统资源在其不再被需要的时候应当显式地被释放。该要求对窗口小部件本身和它们使用过的所有资源（如图形、字体和颜色）都起作用。基本原则是，如果你创建了一个窗口小部件，你必须使用它的dispose()方法销毁该部件。如果你使用任意的系统资源，如系统颜色，那么你不应当释放它们。

幸运的是，窗口小部件的子部件在该窗口小部件被销毁时，也将被自动销毁。这意味着如果你正确地销毁了一个shell，你不再需要销毁它的每一个子部件。这是因为它们将会被自动释放。

4.2.3 窗口小部件事件

事件（event）是当用户执行鼠标或键盘操作时通知应用程序的机制。应用程序可以获得关于输入文本、鼠标点击、鼠标移动、焦点改变等信息。事件由向窗口小部件添加监听器进行处理。比如，SelectionListener用于通知程序用按下并释放了一个Button，或者是在列表中选中了一项。或者，所



图4-3 运行此可以单独运行的SWT程序

有窗口小部件支持Dispose事件，该事件在窗口小部件被销毁之前触发。

对于每一类型的事件，SWT定义了监听器接口（比如，<EventName>Listener）、一个事件类和一个适配器类（如果需要）。请注意适配器类仅当监听器接口定义了多于一个方法时才被提供。此外，对于每一个实现了特定事件的窗口小部件，存在对应的add<EventName>Listener和remove<EventName>Listener方法。

表4-1展示了由SWT定义的事件类型的列表，以及事件生成时的描述，和生成该事件的窗口小部件的列表。

表4-1 窗口小部件事件

事件名称	何时产生	窗口小部件
Arm	一个菜单项准备就绪（高亮显示）	MenuItem
Control	控件被改变大小或移动	Control、TableColumn、Tracker
Dispose	销毁控件	Widget
Focus	控件获取或失去焦点	Control
Help	用户请求帮助（比如，按下F1键）	Control、Menu、MenuItem
Key	按下或释放一个键	Control
Menu	隐藏或显示一个菜单	Menu
Modify	文本被修改	Combo、Text
Mouse	按下、释放或双击鼠标	Control
MouseMove	鼠标移动过控件	Control
MouseTrack	鼠标进入、离开或悬停在控件上	Control
Paint	控件需要重新绘制	Control
Selection	选中控件中的选项	Button、Combo、CoolItem、List、MenuItem、Sash、Scale、ScrollBar、Slider、StyledText、TabFolder、Table、TableColumn、TableTree、Text、ToolItem、Tree
Shell	最小化、最大化、激活、禁用或关闭shell	Shell
Traverse	遍历控件（使用制表键遍历）	Control
Tree	收缩或展开树中的一项	Tree、TableTree
Verify	即将修改文本	Text、StyledText

请注意：此表根据《Platform Plug-in Developer Guide for Eclipse》修改

4.2.4 抽象窗口小部件类

系统中所有的UI对象由抽象类Widget和Control派生而来（图4-4）。本节和下一节讨论了主要小部件类型和它们主要的API。API描述来源于Eclipse平台的Javadoc。

注意 对于每一个事件，都有一个add<EventName>Listener方法和一个对应的remove<EventName>Listener方法。同样地，对于每一个窗口小部件属性，都有一个get<PropertyName>和一个set<PropertyName>方法。为了节省篇幅，仅列出了add<EventName>Listener和set<PropertyName>方法。每一个窗口小部件类型都有一个构造函数。该构造函数需要该窗口小部件的父部件作为第一个参数，样式（一个int类型的变量）作为第二个参数。

1. Widget

Widget类是下列类的抽象超类：Caret、Control（稍后讨论）、DragSource、DropTarget、Item、Menu（在4.2.7节中讨论）、ScrollBar和Tracker。

常用API包括：

- `addDisposeListener(DisposeListener)`——将监听器添加至监听器集。当窗口小部件被释放时将通知该监听器集。
- `addListener(int, Listener)`——将监听器添加至监听器集。当给定类型的事件发生时将通知该监听器集。
- `dispose()`——将与接收者和所有它的后代相关联的系统资源释放。
- `getData(String)`——以指定名称返回程序定义的接收者的属性，如果该属性没有被设置，则返回null。
- `isDisposed()`——如果窗口小部件已经被释放，返回true，否则返回false。
- `notifyListeners(int, Event)`——通过调用`handleEvent()`方法，将该事件的发生通知给接收者的指定类型事件的所有监听器。
- `setData(String, Object)`——使用指定名称和给定值设置程序定义的接收者的属性。
- `toString()`——返回一个包含简明、可读的窗口小部件的描述的字符串。

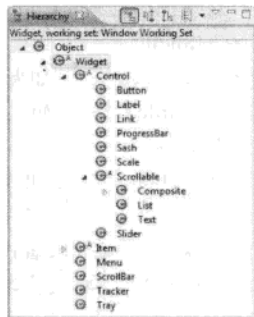


图4-4 SWT窗口小部件层次结构

2. Control

Control类是所有对话框和窗口组件类的抽象超类，如Button、Label、ProgressBar、Sash、Scrollbar和Slider（都将在这一章稍后进行介绍）。常用API包括：

- `addControlListener(ControlListener)`——将监听器添加至监听器集。当控件被移动或改变大小时，通过发送由ControlListener接口定义的消息之一通知该监听器集。
- `addFocusListener(FocusListener)`——将监听器添加至监听器集。当控件获得或失去焦点时，通过发送由FocusListener接口定义的消息之一通知该监听器集。
- `addHelpListener(HelpListener)`——将监听器添加至监听器集。当为控件生成帮助事件时，通过发送由HelpListener接口定义的消息之一通知该监听器集。
- `addKeyListener(KeyListener)`——将监听器添加至监听器集。当键盘的键被按下或释放时，通过发送由KeyListener接口定义的消息之一通知该监听器集。
- `addMouseListener(MouseListener)`——将监听器添加至监听器集。当鼠标按钮被按下或释放时，通过发送由MouseListener接口定义的消息之一通知该监听器集。
- `addMouseMoveListener(MouseMoveListener)`——将监听器添加至监听器集。当鼠标移动时，通过发送由MouseMoveListener接口定义的消息之一通知该监听器集。
- `addMouseTrackListener(MouseTrackListener)`——将监听器添加至监听器集。当鼠标在控件上经过或悬停时，通过发送由MouseTrackListener接口定义的消息之一通知该监听器集。
- `addPaintListener(PaintListener)`——将监听器添加至监听器集。当需要绘制接收者时，通过发送由PaintListener接口定义的消息之一通知该监听器集。
- `addTraverseListener(TraverseListener)`——将监听器添加至监听器集。当遍历事件发生时，通过发送由TraverseListener接口定义的消息之一通知该监听器集。

- `getDisplay()`——返回`display`，接收者创建于该`display`。
- `getParent()`——返回接收者的父部件。当接收者是一个由`null`或父部件的`display`创建的`shell`时，该父部件必须是`Composite`或`null`。
- `getShell()`——返回接收者的`shell`。
- `isDisposed()`——如果窗口小部件被释放了，返回`true`，否则返回`false`。
- `isEnabled()`——如果接收者是可用的，且所有接收者的祖先是可用的，返回`true`。否则返回`false`。
- `isVisible()`——如果接收者是可见的，且所有接收者的祖先也是可见的，返回`true`。否则返回`false`。
- `pack()`——将接收者设为它的默认的大小。
- `redraw()`——将接收者的整个边框标记为需要重新绘制。
- `setBackground(Color)`——将接收者的背景颜色设置为函数的参数。如果该参数是`null`，则设置为默认的系统颜色。
- `setBounds(Rectangle)`——将接收者的大小和位置设置为参数指定的矩形区域。
- `setEnabled(boolean)`——如果参数是`true`，将接收者设置为可用，否则设置为不可用。
- `boolean setFocus()`——使接收者获得键盘焦点，这样所有的键盘事件都将传递至它。
- `setFont(Font)`——设置由参数指定的字体。这些字体将被接收者用来绘制文本信息。如果参数是`null`，则将该控件的字体设置为它的默认字体。
- `setForeground(Color)`——将接收者的前景颜色设置为由参数指定的颜色。如果参数是`null`，则设置为默认系统颜色。
- `setLayoutData(Object)`——将与接收者相关联的布局数据设置为参数指定的数据。
- `setLocation(Point)`——将接收者的父部件的位置设置为参数指定的点（如果它没有父部件，则相对于它的`display`）。
- `setRedraw(boolean)`——如果参数是`false`，将忽略接收者剩下的绘制操作。
- `setSize(Point)`——将接收者的大小设置为参数指定的点。
- `setToolTipText(String)`——将接收者的提示文本设置为参数。参数可以是`null`，表示不显示任何提示。
- `setVisible(boolean)`——如果参数是`true`，将接收者标记为可见。否则标记为不可见。
- `update()`——强迫在本方法返回之前，处理该窗口小部件的所有绘制请求。

3. Scrollable

`Scrollable`类是所有可以拥有滚动条的控件的抽象超类，如`Composite`、`List`和`Text`。常用API包括：

- `getClientArea()`——返回一个描述接收者适于显示数据的区域的矩形（即，没有被“切边”（`trimmings`）覆盖的）。
- `getHorizontalBar()`——如果已有则返回接收者的水平滚动条，否则返回`null`。
- `getVerticalBar()`——如果已有则返回接收者的竖直滚动条，否则返回`null`。

4.2.5 最高级类

如上所述，每一个SWT程序需要一个`display`和一个或多个`shell`（代表每一个窗口帧）。

1. Display

display代表底层系统、UI线程和SWT之间的连接。虽然Display构造函数是公开的，在正常情况下，你不应当创建新的display实例（除非你是在创建一个可单独运行的SWT程序）；作为替代，下列两个静态Display方法返回一个实例。

- `getCurrent()`——返回与当前运行线程相关联的display。如果当前运行线程不是任意display的UI线程，则返回null。
- `getDefault()`——返回默认display。这是由系统初次创建的实例。

对创建窗口小部件或修改当前可见窗口小部件的SWT方法的调用必须由UI线程发出；否则，将抛出一个`SWTException`，表示调用由非UI线程发出。对前面列出的`getCurrent()`方法可被用于快速确定当前线程是UI还是非UI线程。如果当前线程是非UI的，下列Display方法可以用来在下一个可用时间在UI线程上请求执行。

- `asyncExec(Runnable)`——使runnable的`run()`方法在下一个合理的时机由UI线程调用，且不阻塞调用线程。
- `syncExec(Runnable)`——使runnable的`run()`方法在下一个合理的时机由UI线程调用。调用线程将被阻塞，直到`run()`方法完成执行。
- `timerExec(int, Runnable)`——使runnable的`run()`被UI线程阻塞，直到指定数目的毫秒数过去。

这些方法，和上面列出的方法，可以用来对资源更改事件做出响应时更新可见的窗口小部件（请查看9.2节）、显示错误消息（请查看21.4.3节）或简单延后执行，直到窗口小部件被初始化（请查看8.2.5节）。

除了管理UI事件循环之外，它还提供了对SWT所需的系统资源的访问方法。常用API包括：

- `addListener(int, Listener)`——将监听器添加至监听器集。当给定类型的事件发生时将通知该监听器集。
- `beep()`——使系统硬件发出短鸣声（如果硬件支持该功能）。
- `close()`——请求SWT和底层操作系统间的连接将被关闭。
- `disposeExec(Runnable)`——当display被释放时，使runnable的`run()`方法被UI线程触发。
- `findWidget(int)`——对于给定窗口小部件的操作系统句柄，返回Widget子类的实例。该子类在当前运行的程序中，如果实例存在，就代表该实例。如果无法找到对应窗口小部件，将是null。
- `getActiveShell()`——返回当前活动Shell，如果当前运行程序没有所属的shell是活动的，则返回null。
- `getBounds()`——返回一个描述接收者的大小和位置的矩形。
- `getClientArea()`——返回一个描述接收者中适合显示数据的区域的矩形。
- `getCursorControl()`——返回当前屏幕上指针悬停的控件。如果指针没有悬停于当前运行程序的任意控件，则返回null。
- `getCursorLocation()`——返回当前屏幕上的指针相对于屏幕左上角的位置。
- `getData(String)`——返回由程序定义的指定名称的接收者的属性。如果没有被设置，则返回null。
- `getDoubleClickTime()`——返回底层操作系统可被认为是鼠标双击的两次鼠标点击间最长的时间间隔，以毫秒的形式。
- `getFocusControl()`——返回当前拥有键盘焦点的控件。如果键盘事件不传递至当前运行程序的任意控件，则返回null。
- `getShells()`——返回一个包含所有未被释放的shell的数组。并将接收者作为它们的display。

- `getSystemColor(int)`——返回给定常量的对应标准颜色。该给定常量应是由SWT类定义的颜色常量之一。
- `getSystemFont()`——返回一个合理的字体以供程序使用。
- `readAndDispatch()`——从操作系统事件队列读取一个事件，并恰当的分发。如果还有更多工作要做，返回true。如果调用者在另一个事件被放置于事件队列之前可以睡眠，则返回false。
- `setCursorLocation(Point)`——设置屏幕指针的相对于屏幕左上角的相对位置。
- `setData(String, Object)`——用给定参数的指定名称设置接收者的程序定义属性。
- `sleep()`——使UI线程睡眠（即处于不消耗CPU周期状态中），直至一个事件被接受或线程被唤醒。
- `update()`——强制display的所有绘制请求在该方法返回前完成。

2. Shell

每个窗口都有一个shell，代表用户可以与之交互的窗口帧。shell处理对所有窗口通用的移动和改变大小的行为，它也是在它边框内显示的所有窗口小部件的父部件（参见11.1.10节）。常用API包括：

- `addShellListener(ShellListener)`——将监听器添加至监听器集。当在接收者上完成操作时，通过传递由ShellListener接口定义的消息之一，通知该监听器集。
- `close()`——请求窗口管理器关闭接收者，以与用户点击“关闭框”和输入其他系统定义的代表窗口将被移除的键或鼠标组合操作同样的方式关闭。
- `dispose()`——释放与接收者及其所有后代相关联的操作系统资源。
- `getDisplay()`——返回创建接收者的display。
- `getShell()`——返回接收者的shell。
- `getShells()`——返回一个包含接收者的后代shell的数组。
- `isEnabled()`——如果接收者及其所有祖先可用，返回true；否则返回false。
- `open()`——将接收者移至创建它的display的绘制顺序的顶部（以使所有在该display上面的不是接收者的后代shell将会在它后面绘制），将它标记为可见。将焦点设置它的默认按钮（如果已有），并请求窗口管理器使该shell成为活动的。
- `setActive()`——将接收者移至创建它的display的绘制顺序的顶部（以使所有在该display上的，不是接收者的后代的shell都将在它后面绘制），并请求窗口管理器将该shell设为活动的。
- `setEnabled(boolean enabled)`——如果参数的值是true，将接收者设为可用。否则，设为不可用。
- `setVisible(boolean visible)`——如果参数值是true，将接收者设为可见；否则，设为不可见。

4.2.6 常用窗口小部件

在SWT类层次结构中定义了数十种窗口小部件。本节讨论了在插件开发过程中最常用的窗口小部件，如标签、按钮、文本字段、列表、表、树、容器和选项卡文件夹（tab folder）。同时也为每一个小部件提供了常用API列表和创建样式。

1. 标签

标签是显示字符串或图像作为其内容的静态控件。它们不生成任意特定的事件，而且也不支持任何用户交互。常用API包括：

- `setAlignment(int)`——控制在接收者中如何显示文本和图像。合格的参数包括SWT.LEFT、SWT.RIGHT和SWT.CENTER。
- `setImage(Image)`——将接收者的图像设置为参数。如果参数为null，则表示不显示任何图像。

- `setText(String)`——设置接收者的文本。

有用的创建样式包括：

- `SWT.SHADOW_IN`——在窗口小部件周围创建嵌入阴影。
- `SWT.SHADOW_OUT`——在窗口小部件周围创建浮动阴影。
- `SWT.SHADOW_NONE`——创建一个没有阴影的窗口小部件。
- `SWT.WRAP`——将窗口小部件的文本在必要时转换为多行。
- `SWT.SEPARATOR`——创建一个竖直或水平的线。
- `SWT.HORIZONTAL`——创建一条水平线。
- `SWT.VERTICAL`——创建一条竖直线。
- `SWT.LEFT`——在窗口小部件的边框内使其左对齐。
- `SWT.RIGHT`——在窗口小部件的边框内使其右对齐。
- `SWT.CENTER`——在窗口小部件的边框内使其居中对齐。

2. 按钮

按钮提供了当被点击时初始化操作的机制。当被按下和释放时，它们将产生一个Selection事件。按钮可以显示字符串或图像作为它们的内容。取决于它们的样式设定，按钮可以代表一些普通的UI元素类型，如按钮、单选框、单选按钮、切换按钮和箭头按钮。常用API包括：

- `addSelectionListener(SelectionListener)`——将监听器添加至监听器集。当控件被选中时，通过传递由SelectionListener接口定义的消息之一，通知该监听器集。
- `getSelection()`——如果接收者被选中，返回true；否则返回false。
- `setAlignment(int)`——控制在接收者如何显示文本、图像和箭头。
- `setImage(Image)`——将接收者的图像设置为参数的值。如果是null，则表示不显示任何图像。
- `setSelection(boolean)`——如果接收者的类型是`SWT.CHECK`，`SWT.RADIO`，或`SWT.TOGGLE`，则设置接收者的选择状态。
- `setText(String)`——设置接收者的文本。

有用的创建样式有：

- `SWT.ARROW`——创建一个箭头按钮。
- `SWT.CHECK`——创建一个单选框。
- `SWT.PUSH`——创建一个按钮。
- `SWT.RADIO`——创建一个单选按钮。
- `SWT.TOGGLE`——创建一个切换按钮。
- `SWT.UP`——创建一个向上箭头按钮。
- `SWT.DOWN`——创建一个向下箭头按钮。
- `SWT.LEFT`——创建一个向左箭头按钮，或将窗口小部件在它的边框内左对齐。
- `SWT.RIGHT`——创建一个向右箭头按钮，或将窗口小部件在它的边框内右对齐。
- `SWT.CENTER`——将窗口小部件在它的边框内居中对齐。

下列示例代码（没有包声明语句的）创建了一个带有单一按钮的窗口。点击按钮将会改变按钮的文本（图4-5）。

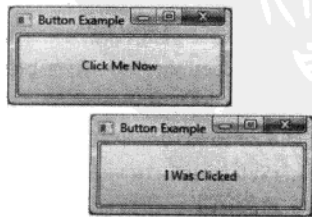


图4-5 按钮示例

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class ButtonExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Button Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final Button button = new Button(shell, SWT.PUSH);
        button.setText("Click Me Now");

        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                button.setText("I Was Clicked");
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

相对于本章的第一个示例，在上面示例中加粗显示重要的行。在创建了按钮后，添加一个选择监听器，并给该监听器创建了一个覆盖widgetSelected()方法的SelectionAdapter。

3. 文本

文本窗口小部件提供了文本查看和编辑功能。如果用户输入了超过窗口小部件可以显示的长度的文本，文本将会自动滚动。常用API包括：

- addModifyListener(ModifyListener)——将监听器添加至监听器集。当接收者的文本被更改时，通过传递由ModifyListener接口定义的消息之一，通知该监听器集。
- addSelectionListener(SelectionListener)——将监听器添加至监听器集。当控件被选中时，通过传递由SelectionListener接口定义的消息之一，通知该监听器集。
- addVerifyListener(VerifyListener)——将监听器添加至监听器集。当接收者的文本被验证时，通过传递由VerifyListener接口定义的消息之一，通知该监听器集。在用户完成一些任务以修改文本（一般地，输入一个键）后，但在文本被修改之前，发生验证事件。
- clearSelection()——清除选择。
- copy()——复制选中文本。
- cut()——剪切选中文本。
- getSelectionText()——获取选中文本。
- getText()——获取窗口小部件的文本。
- getText(int start, int end)——获取一个范围的文本。

- `insert(String)`——插入一个文本。
- `paste()`——从剪贴板粘贴文本。
- `selectAll()`——选择接收者中所有文本。
- `setEchoChar(char echo)`——设置回显字符。
- `setEditable(boolean editable)`——设置可编辑对象的状态。
- `setSelection(int start, int end)`——设置选择。
- `setText(String)`——将接收者的内容设为参数的值。
- `setTextLimit(int)`——设置接收者有能力容纳的字符的最大个数为参数的值。
- `setTopIndex(int)`——设置当前接收者顶部的行的相对于0的索引。

有用的创建样式包括：

- `SWT.SINGLE`——创建一个单行文本窗口小部件。
- `SWT.MULTI`——创建一个多行文本窗口小部件。
- `SWT.WRAP`——使部件的文本在需要时转换为多行。
- `SWT.READ_ONLY`——创建一个不能更改的只读文本窗口小部件。
- `SWT.LEFT`——创建一个左对齐的文本窗口小部件。
- `SWT.RIGHT`——创建一个右对齐的文本窗口小部件。
- `SWT.CENTER`——创建一个居中对齐的文本窗口小部件。

以下示例代码创建了包含一个单一行文本字段的窗口帧。该文本只接受数字输入（图4-6）。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TextExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Text Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final Text text = new Text(shell, SWT.MULTI);
        text.addVerifyListener(new VerifyListener() {
            public void verifyText(VerifyEvent event) {
                event.doit = event.text.length() == 0
                    || Character.isDigit(event.text.charAt(0));
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

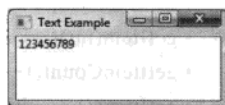


图4-6 文本示例

和上一个示例一样，重要的行以加粗形式高亮显示。在创建了文本窗口小部件之后，添加一个

验证监听器。在该监听器中，通过覆盖verifyText()方法创建了一个VerifyListener，以用于验证输入字符是否是数字。请注意：如果用户删除或撤销一些文本，则event.text将为空。

4. 列表

列表窗口小部件表示一个包含项目的列表，它允许用户选择一个或多个项目。当选中一项时，列表生成一个Selection事件。常用API包括：

- add(String)——将参数添加至接收者列表的末端。
- addSelectionListener(SelectionListener)——将监听器添加至监听器集。当接收者的选择改变时，通过传递由SelectionListener定义的消息之一，通知该监听器集。
- deselect(int)——取消选中接收者的给定的相对于0的索引的项目。
- deselectAll()——取消选中接收者的所有被选中项目。
- getItem(int)——返回接收者中，给定的对于0的索引位置的项目。
- getItemCount()——返回接收者包含的项目的个数。
- getItems()——返回一个包含接收者的项目的数组。
- getSelection()——返回一个包含接收者当前被选中的项目的数组。
- getSelectionCount()——返回接收者包含的被选中的项目的数量。
- getSelectionIndex()——返回接收者中当前被选中的项目的相对于0的索引，如果没有被选中项目，则返回-1。
- getSelectionIndices()——返回接收者中当前被选中的所有项目的相对于0的索引。
- indexOf(String)——获取一个项目的索引。
- remove(int)——从接收者中删除相对于0的指定索引位置的项目。
- remove(String)——从第一个项目开始搜索接收者的列表直到等于参数的项目被找到，并从列表中删除该项目。
- removeAll()——从接收者删除所有项目。
- select(int)——从接收者列表中选择给定的，相对于0的索引位置的项目。
- selectAll()——选择接收者中的所有项目。
- setItems(String[] items)——将接收者的项目设置为给定项目数组。
- setSelection(int)——从接收者中选择给定的，相对于0的索引位置的项目。
- setSelection(String[])——将接收者的选择设为给定的项目数组。
- setTopIndex(int)——设置接收者当前顶部的行的相对于0的索引位置。

有用的创建样式包括：

- SWT.SINGLE——创建一个单一选项的列表窗口小部件。
- SWT.MULTI——创建一个多选项的列表窗口小部件。

随后的示例使用一个单选列表框创建了一个窗口帧。在一个项目上单击或双击将向终端输出该选择（图4-7）。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class ListExample {
```

```
public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("List Example");
    shell.setBounds(100, 100, 200, 100);
    shell.setLayout(new FillLayout());
    final List list = new List(shell, SWT.SINGLE);
    list.setItems(new String []
        {"First", "Second", "Third"});
    list.addSelectionListener(new SelectionAdapter(){
        public void widgetSelected(SelectionEvent event){
            String [ ] selected =list.getSelection();
            if (selected.length >0)
                System.out.println(
                    "Selected:"+selected [0 ]);
        }
        public void widgetDefaultSelected(
            SelectionEvent event){
            String [ ] selected =list.getSelection();
            if (selected.length >0)
                System.out.println(
                    "Default Selected:"+selected [0 ]);
        }
    });
    shell.open();
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) display.sleep();
    }
    display.dispose();
}
```

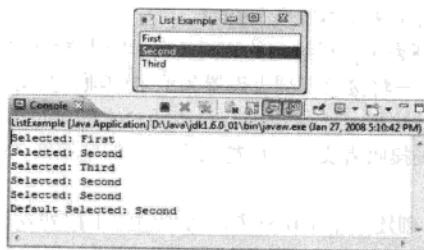


图4-7 列表示例

在列表窗口小部件创建后，它的内容通过setItems()方法设置。然后，添加一个选择监听器，在该监听器中通过覆盖widgetSelected()和widgetDefaultSelected()方法创建了一个SelectionAdapter，用来打印任何被选中或双击的项目。

5. 组合框

与列表窗口小部件类似，组合框窗口小部件允许用户从一个可用项目列表中选择一个项目。决

定于组合框是如何被设置，它可能也允许用户在文本字段中输入一个新的值。最后被选中或输入的项目显示在文本框中。常用API包括：

- `add(String)`——将参数添加至接收者的列表末端。
- `addModifyListener(ModifyListener)`——将监听器添加至监听器集。当接收者的文本被修改时，通过传递由`ModifyListener`接口定义的消息之一，通知该监听器集。
- `addSelectionListener(SelectionListener)`——将监听器添加至监听器集。当接收者的选择被改变时，通过传递由`SelectionListener`接口定义的消息之一，通知该监听器集。
- `clearSelection()`——将接收者的文本字段的选中设置为一个从第一个字符开始的空的选择。
- `copy()`——复制选中文本。
- `cut()`——剪切选中文本。
- `deselect(int)`——取消选中接收者列表中的，给定的相对于0的索引位置的项目。
- `deselectAll()`——取消选中接收者列表中的所有被选中项目。
- `getItem(int)`——返回接收者列表中的、给定的相对于0的索引位置的项目。
- `getItemCount()`——返回接收者列表包含的项目的数目。
- `getItems()`——返回一个包含接收者列表中的所有项目的数组。
- `getSelectionIndex()`——返回接收者列表中当前被选中项的相对于0的索引。如果没有项目被选中，则返回-1。
- `getText()`——返回包含接收者文本字段的内容的复制内容的字符串。
- `indexOf(String)`——从第一个项目（索引0）开始搜索接收者列表，直到等于参数的项目被找到，并返回该项目的索引。
- `paste()`——从剪贴板粘贴文本。
- `remove(int)`——从接收者列表删除给定的、相对于0的索引位置的项目。
- `remove(String)`——从第一个项目开始搜索接收者的列表，直到一个等于参数的项目被找到，并从列表中删除该项目。
- `removeAll()`——从接收者列表中删除所有项目。
- `select(int)`——在接收者列表中选择给定的相对于0的索引位置的项目。
- `setItems(String[] items)`——将接收者列表设置为给定的项目数组。
- `setText(String)`——将接收者的文本字段内容设置为给定的字符串。
- `setTextLimit(int)`——设置接收者文本字段能最大容纳的字符数为参数的值。

有用的创建样式包括：

- `SWT.DROP_DOWN`——创建一个下拉列表。可编辑的下拉列表也被称为复选框。
- `SWT.READ_ONLY`——创建一个只读的下拉列表。
- `SWT.SIMPLE`——创建一个总是显示列表的组合框。

以下示例创建了一个具有两个复选框和一个标签的窗口帧。从第一个、第二个复选框中选择一个项目或在第二个复选框中输入一个新的值将会改变标签的内容以反映该选择（图4-8）。

```
import org.eclipse.swt.*;  
import org.eclipse.swt.events.*;  
import org.eclipse.swt.layout.*;
```

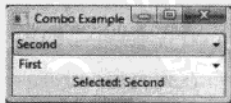


图4-8 复选框示例

```
import org.eclipse.swt.widgets.*;

public class ComboExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Combo Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout(SWT.VERTICAL));
        final Combo combo1 = new Combo(shell, SWT.READ_ONLY);
        final Combo combo2 = new Combo(shell, SWT.DROP_DOWN);
        final Label label = new Label(shell, SWT.CENTER);
        combo1.setItems(new String [ ]
            { "First", "Second", "Third" });
        combo1.setText("First");
        combo1.addSelectionListener(new SelectionAdapter(){
            public void widgetSelected(SelectionEvent event){
                label.setText("Selected: "+combo1.getText());
            }
        });
        combo2.setItems(new String [ ]
            { "First", "Second", "Third" });
        combo2.setText("First");
        combo2.addModifyListener(new ModifyListener(){
            public void modifyText(ModifyEvent event){
                label.setText("Entered: "+combo2.getText());
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

在创建复选框和标签之后，由setItems()方法设置复选框的内容，由setText()方法设置它们的初始选择（文本字段的内容）。一个选件监听器被添加至第一个复选框。在该复选框中，通过覆盖widgetSelected()方法创建了一个SelectionAdapter。然后将一个修改监听器添加至第二个复选框。在该复选框中，通过覆盖modifyText()方法创建了一个ModifyListener。当它们对应的复选框改变选择时，这两个方法均更新了标签的内容。

6. 表

表窗口小部件提供了一个竖直的、多列的项目列表。它可以在列表中以一行内容显示每一个项目。表的列由一个或多个TableColumn实例定义。每一个定义它自己的头部、宽度和对齐方式。常用API包括：

- addSelectionListener(SelectionListener)——将监听器添加至监听器集。当接收者的选择改变时，通过传递由SelectionListener接口定义的消息之一通知该监听器集。
- deselect(int)——取消选中接收者中给定的相对于0的索引位置的项目。

- `deselectAll()`——取消选中接收者中所有被选中项目。
- `getColumn(int)`——返回接收者的给定的相对于0的索引位置的列。
- `getColumns()`——返回一个`TableColumns`类型的数组。该数组包含了接收者中的列。
- `getItem(int)`——返回接收者中给定的相对于0的索引位置的项目。
- `getSelection()`——返回一个`TableItems`类型的数组。该数组包含在接收者中被选中的项目。
- `getSelectionCount()`——返回接收者包含的被选中项目的数目。
- `getSelectionIndex()`——返回接收者当前被选中的项目的相对于0的索引。如果没有项目被选中，则返回-1。
- `getSelectionIndices()`——返回接收者中当前被选中项目的相对于0的所有索引。
- `indexOf(TableColumn)`——从第一列（索引0）开始搜索接收者的列表，直到一个与参数相等的项目被找到。返回该项目的索引。
- `indexOf(TableItem)`——从第一项（索引0）开始搜索接收者的列表，直到一个与参数相等的项目被找到。返回该项目的索引。
- `remove(int)`——从接收者中删除给定的相对于0的索引位置的项目。
- `removeAll()`——从接收者删除所有项目。
- `select(int)`——从接收者中选择给定的相对于0的索引位置的项目。
- `selectAll()`——选择接收者中的所有项目。
- `setHeaderVisible(boolean)`——如果参数是`true`，则将接收者的头部标记为可见。否则，标记为不可见。
- `setLinesVisible(boolean)`——如果参数为`true`，则将接收者的行标记为可见。否则，标记为不可见。
- `setSelection(int)`——选择接收者的给定的相对于0的索引位置的项目。
- `setSelection(TableItem[])`——将接收者的选项设置为给定的项目数组。
- `setTopIndex(int)`——将接收者当前顶部项目的相对于0的索引设置为参数的值。

有用的创建样式包括：

- `SWT.SINGLE`——创建一个单选项的表。
- `SWT.MULTI`——创建一个多选项的表。
- `SWT.CHECK`——创建一个单选框表。
- `SWT.FULL_SELECTION`——创建一个可以选中行（而不是选中单元格）的表。

有用的`TableColumn`的API包括：

- `addControlListener(ControlListener)`——将监听器添加至监听器集。当控件被移动或改变大小时，通过传递由`ControlListener`接口定义的消息之一，通知该监听器集。
- `addSelectionListener(SelectionListener)`——将监听器添加至监听器集。当控件被选中时，通过传递由`SelectionListener`接口定义的消息之一，通知该监听器集。
- `pack()`——使接收者变为它的首选大小。
- `setAlignment(int)`——控制接收者中文本和图像的显示方式。
- `setImage(Image)`——设置接收者的图像为参数。如果参数为`null`，表示不显示图像。
- `setResizable(boolean)`——设置可改变大小的属性。
- `setText(String)`——设置接收者的文本。
- `setWidth(int)`——设置接收者的宽度。

- 有用的TableItem API包括:
- `getChecked()`——如果接收者被选中, 返回true。否则返回false。
- `getText(int)`——返回接收者中的, 给定栏索引中存储的文本。如果文本未被设置, 则返回空字符串。
- `setBackground(Color)`——将接收者的背景颜色设置为参数指定的颜色。如果参数是null, 则将设为默认的系统颜色。
- `setChecked(boolean)`——设置该项的单选框的确认状态。
- `setForeground(Color)`——设置接收者的前景颜色为参数指定的颜色。如果参数是null, 则设置为默认系统颜色。
- `setGrayed(boolean)`——设置该项的单选框的无效状态。
- `setImage(Image)`——设置接收者的图像为参数。如果是null, 表示不显示图像。
- `setImage(Image[])`——设置表中多栏的图像。
- `setImage(int, Image)`——设置接收者某一栏的图像。
- `setImageIndent(int)`——设置图像缩进。
- `setText(int, String)`——设置接收者某一栏的文本。
- `setText(String)`——设置接收者的文本。
- `setText(String[])`——设置表中多栏的文本。

下面的示例创建了一个两栏, 两项的表。在一个项目上点击将把该单元格的内容输出至终端(图4-9)。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TableExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Table Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final Table table = new Table(shell,
            SWT.SINGLE | SWT.BORDER | SWT.FULL_SELECTION);
        table.setHeaderVisible(true);
        table.setLinesVisible(true);
        TableColumn column1 =
            new TableColumn(table, SWT.NULL);
        column1.setText("Name");
        column1.pack();
        TableColumn column2 =
            new TableColumn(table, SWT.NULL);
        column2.setText("Age");
        column2.pack();
        TableItem item1 = new TableItem(table, SWT.NULL);
        item1.setText(new String[] { "Dan", "43" });
        TableItem item2 = new TableItem(table, SWT.NULL);
```

```

        item2.setText(new String [ ] {"Eric", "44"});
        table.addSelectionListener(new SelectionAdapter(){
            public void widgetSelected(SelectionEvent event){
                TableItem [] selected =table.getSelection();
                if (selected.length >0){
                    System.out.println("Name:" +
                        selected [0 ].getText(0));
                    System.out.println("Age:" +
                        selected [0 ].getText(1));
                }
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}

```

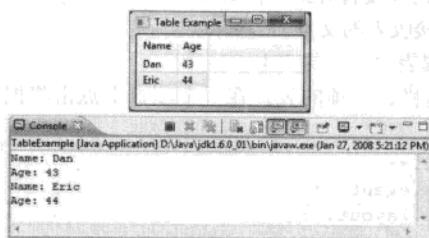


图4-9 表示例

创建的该表具有全选功能。通过setHeaderVisble()方法将表题设为可见，通过setLinesVisible()方法将它的行设为可见。然后，创建每一栏，并通过setText()方法设置它的栏的标题。pack()方法设置了每一栏的大小为它的内容的最大值。然后，创建了每一个表的行项目，并通过setText()（预期参数为一个字符串数组，每一个字符串对应于一栏）方法设置单元格内容。最后，给表添加了一个选择监听器。在该监听器通过覆盖widgetSelected(方法中创建了一个SelectionAdapter以输出任意被选中项目。

7. 树

树窗口小部件用于以层次结构的方式显示信息。树由一个项目列表组成。这些项目可以由其他项目组成。这些其他项目还可以由其他项目组成等。用户通过展开与收缩项目在树中浏览，查看项目并隐藏它们的组成项。常用API包括：

- addSelectionListener(SelectionListener)——将监听器添加至监听器集。当接收者的选择改变时，通过传递由SelectionListener接口定义的消息之一通知该监听器集。
- addTreeListener(TreeListener)——将监听器添加至监听器集。当接收者被展开或收缩时，通过传递由TreeListener接口定义的消息之一通知该监听器集。
- deselectAll()——取消选中接收者中所有被选中项目。

- `getItemCount()`——返回接收者中属于接收者直接子项目的项目个数。
- `getItems()`——返回接收者中属于接收者直接子项目的项目。
- `getSelection()`——返回一个接收者当前被选中的`TreeItems`数组。
- `getSelectionCount()`——返回接收者中包含的被选中项目的个数。
- `removeAll()`——移除接收者中的所有项目。
- `selectAll()`——选择接收者中的所有项目。
- `setSelection(TreeItem[])`——设置接收者的选择为给定的项目数组。
- `setTopItem(TreeItem)`——设置当前接收者中的顶部项目。

有用的创建样式包括:

- `SWT.SINGLE`——创建一棵单选树。
- `SWT.MULTI`——创建一棵多选树。
- `SWT.CHECK`——创建一棵单选框树。

有用的`TreeItem` API包括:

- `getChecked()`——如果接收者被确认, 返回`true`; 否则, 返回`false`。
- `getExpanded()`——如果接收者被展开, 返回`true`; 否则, 返回`false`。
- `getItemCount()`——返回接收者中属于接收者直接子项目的项目个数。
- `getItems()`——返回一个包含接收者直接子项目的`TreeItems`数组。
- `getParent()`——返回接收者的父部件, 它必须是一棵树。
- `getParentItem()`——返回接收者的父项目, 它必须是一个`TreeItem`。如果接收者是根, 则返回`null`。
- `setBackground(Color)`——设置接收者的背景颜色为参数指定的颜色。如果参数是`null`, 则设为项目的默认系统颜色。
- `setChecked(boolean)`——设置接收者的确认状态。
- `setExpanded(boolean)`——设置接收者的展开状态。
- `setForeground(Color)`——设置接收者的前景颜色为参数指定的颜色。如果参数是`null`, 则设为项目的默认系统颜色。
- `setGrayed(boolean grayed)`——设置接收者的无效状态。
- `setImage(Image)`——设置接收者的图像为参数。如果参数为`null`, 则表示不显示图像。
- `setText(String)`——设置接收者的文本。

下面的示例创建一个包含三级项目的树。点击一个项目将输出它的名字至终端。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TreeExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Tree Example");
        shell.setBounds(100, 100, 200, 200);
        shell.setLayout(new FillLayout());
        final Tree tree = new Tree(shell, SWT.SINGLE);
```



```

for (int i =1;i <4;i++){
    TreeItem grandParent =new TreeItem(tree,0);
    grandParent.setText("Grand Parent -"+i);
    for (int j =1;j <4;j++){
        TreeItem parent =new TreeItem(grandParent,0);
        parent.setText("Parent -"+j);
        for (int k =1;k <4;k++){
            TreeItem child =new TreeItem(parent,0);
            child.setText("Child -"+k);
        }
    }
}
tree.addSelectionListener(new SelectionAdapter(){
    public void widgetSelected(SelectionEvent event){
        TreeItem [ ] selected =tree.getSelection();
        if (selected.length >0){
            System.out.println("Selected:"+
                selected [0 ].getText());
        }
    }
});
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()) display.sleep();
}
display.dispose();
}
}

```

树创建了之后，将创建新项目，并通过setText()方法设置它们的标签。许多项目具有它们自己的子项目。最后，添加了一个选择监听器。在该监听器上通过覆盖widgetSelected()方法创建了一个SelectionAdapter以输出被选中项目。

8. 复合部件

复合部件 (Composite) 是作为其他窗口小部件的容器。复合部件的后代是包含在它边框内的窗口小部件，并根据复合部件调整自身的大小。常用API包括：

- getChildren()——返回一个包含接收者后代的数组。
- layout()——如果接收者拥有布局，它将请求布局设置接收者的后代的大小和位置。
- setLayout(Layout)——将接收者关联的布局设置为参数，参数可以是null。
- setTabList(Control[])——为指定的控件设置Tab键顺序，该顺序由参数列表中它们出现的先后顺序决定。

有用的创建样式包括：

- SWT.BORDER——创建一个具有边框的复合部件。
- SWT.NO_RADIO_GROUP——组织子单选按钮的行为。
- SWT.H_SCROLL——创建一个具有水平滚动条的复合部件。

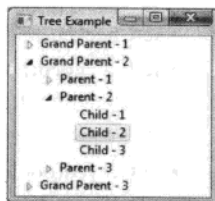


图4-10 树示例

• **SWT.V_SCROLL**——创建一个具有竖直滚动条的复合部件。

以下示例扩展了之前的按钮示例，在shell和按钮间插入一个复合部件（图4-11）。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class CompositeExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Composite Example");
        shell.setBounds(100, 100, 200, 200);
        Composite composite = new Composite(
            shell, SWT.BORDER);
        composite.setBounds(25, 25, 150, 125);
        final Button button = new Button(composite, SWT.PUSH);
        button.setBounds(25, 25, 100, 75);
        button.setText("Click Me Now");
        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                button.setText("I Was Clicked");
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

一个复合部件被作为shell的子部件创建，然后复合部件作为按钮的父部件发挥作用。请注意按钮是相对于复合部件放置，而不是shell。

9. 组

组（Group）窗口小部件是一种特殊的复合部件。它用蚀刻的边框和可选的标签包围子部件。每一个子部件包含在组的边框内，并相对于组自我调整大小。常用API包括：

- **getChildren()**——返回一个包含接收者子部件的数组。
- **layout()**——如果接收者具有布局，它请求布局设置接收者子部件的大小和位置。
- **setLayout(Layout)**——设置接收者关联的布局为参数，参数可以为null。
- **setTabList(Control[])**——设置指定控件的Tab键顺序。该顺序由这些控件在参数列表中出现的先后顺序决定。
- **setText(String)**——设置接收者的文本为参数，该文本将会在接受的标题栏显示。该文本不能为null。

有用的创建样式包括：

- **SWT.BORDER**——创建一个带边框的复合部件。

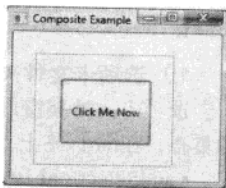


图4-11 复合部件示例

- `SWT.NO_RADIO_GROUP`——阻止子单选按钮的行为。

下列示例代码用一个组部件替代了上面示例中复合部件（图4-12）。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class GroupExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Group Example");
        shell.setBounds(100, 100, 200, 200);
        Group group = new Group(shell, SWT.NULL);
        group.setText("My Group");
        group.setBounds(25, 25, 150, 125);
        final Button button = new Button(group, SWT.PUSH);
        button.setBounds(25, 25, 100, 75);
        button.setText("Click Me Now");
        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                button.setText("I Was Clicked");
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

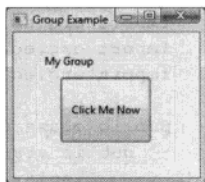


图4-12 组部件示例

10. 选项卡文件夹

选项卡文件夹窗口小部件被用于在一个窗口帧内组织信息为多个页面，以成为一个笔记本标签的集合。在tab标签上点击将使该页前端显示。Tab标签可以带有图像和文本。常用API包括：

- `addSelectionListener(SelectionListener)`——将监听器添加至监听器集。当接收者的选择改变时，通过传递由`SelectionListener`接口定义的消息之一，以通知该监听器集。
- `TabItem getItem(int)`——返回接收者中的给定的相对于0的索引位置的项目。
- `getItemCount()`——返回接收者中包含的项目个数。
- `getItems()`——返回一个包含接收者中的项目的`TabItems`数组。
- `getSelection()`——返回一个接收者中当前选中的`TabItems`数组。
- `getSelectionIndex()`——返回接收者当前被选中的项目相对于0的索引。如果没有项目被选中，则返回-1。
- `indexOf(TabItem item)`——从第一个项目（索引0）开始搜索接收者的列表，直到一个与参数相等的项目被找到，并返回该项目的索引。
- `setSelection(int)`——在接收者选择给定的相对于0的索引位置的项目。

有用的选项卡选项卡API包括：

- `getControl()`——返回当用户选中标签项时，用于该选项卡文件夹中填充客户区域的控件。
- `setControl(Control control)`——设置当用户选中标签项时，用于该选项卡文件夹中填充客户区域的控件。
- `setImage(Image)`——设置接收者的图像为参数。该参数可以为null，表示不显示图像。
- `setText(String)`——设置接收者的文本。
- `setToolTipText(String)`——设置接收者的提示文本为参数，该参数可以是null，表示不显示提示文本。

下面的示例代码创建了一个带有几个选项卡的选项卡文件夹。每一个选项卡包含一个具有单个按钮的复合部件（图4-13）。

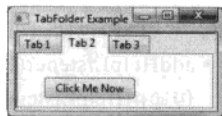


图4-13 选项卡文件夹示例

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TabFolderExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("TabFolder Example");
        shell.setBounds(100, 100, 175, 125);
        shell.setLayout(new FillLayout());
        final TabFolder tabFolder =
            new TabFolder(shell, SWT.BORDER);
        for (int i = 1; i < 4; i++) {
            TabItem tabItem =
                new TabItem(tabFolder, SWT.NULL);
            tabItem.setText("Tab " + i);
            Composite composite =
                new Composite(tabFolder, SWT.NULL);
            tabItem.setControl(composite);
            Button button = new Button(composite, SWT.PUSH);
            button.setBounds(25, 25, 100, 25);
            button.setText("Click Me Now");
            button.addSelectionListener(
                new SelectionAdapter() {
                    public void widgetSelected(
                        SelectionEvent event) {
                        ((Button) event.widget)
                            .setText("I Was Clicked");
                    }
                });
        }
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

当选项卡文件夹被创建后，添加了几个选项卡。对于每个选项卡项目，`setControl()`方法用于用一个复合部件填充它的客户区域。然后，给每一个复合部件添加了一个按钮部件。

4.2.7 菜单

菜单提供了一种供用户触发不同命令和操作的简易方法。最高级菜单可以包含任意数量的子菜单项。常用API包括以下内容：

- `addHelpListener(HelpListener)`——将监听器添加至监听器集。当控件生成帮助事件时，通过传递由`HelpListener`接口定义的消息之一，通知该监听器集。
- `addMenuListener(MenuListener)`——将监听器添加至监听器集。当菜单被隐藏或显示时，通过传递由`MenuListener`接口定义的消息之一，通知该监听器集。
- `getItem(int)`——返回接收者中的给定的相对于0的索引位置的项目。
- `getItemCount()`——返回接收者中包含的项目个数。
- `getItems()`——返回一个包含接收者中的项目的菜单项数组。
- `getParentItem()`——返回接收者的父项目。当接收者是根时，该父项目必须是菜单项或`null`。
- `getParentMenu()`——返回接收者的父项目。当接收者是根时，该父项目必须是菜单或`null`。
- `indexOf(MenuItem item)`——从第一项（索引为0）开始搜索接收者的列表，直到找到一个与参数相同的项目，并返回该项目的索引。
- `setEnabled(boolean enabled)`——如果参数是`true`，将接收者设为可用。否则，设为不可用。
- `setVisible(boolean visible)`——如果参数是`true`，将接收者设为可见。否则，设为不可见。

有用的菜单创建样式包括：

- `SWT.BAR`——创建一个菜单栏。
- `SWT.DROP_DOWN`——创建一个下拉菜单。
- `SWT.POP_UP`——创建一个弹出菜单。

有用的菜单项API包括：

- `addArmListener(ArmListener)`——将监听器添加至监听器集。当控件生成Arm事件时，通过传递由`ArmListener`接口定义的消息之一，通知该监听器集。
- `addHelpListener(HelpListener)`——将监听器添加至监听器集。当控件生成帮助事件时，通过传递由`HelpListener`接口定义的消息之一，通知该监听器集。
- `addSelectionListener(SelectionListener)`——将监听器添加至监听器集。当控件被选中时，通过传递由`SelectionListener`接口定义的消息之一，通知该监听器集。
- `getParent()`——返回接收者的父部件，该父部件必须是菜单。
- `getSelection()`——如果接收者被选中，返回`true`；否则返回`false`。
- `isEnabled()`——如果接收者和它所有祖先是可用的，返回`true`；否则返回`false`。
- `setAccelerator(int accelerator)`——设置部件加速器。
- `setEnabled(boolean enabled)`——如果参数为`true`，将接收者设为可用；否则设为不可用。
- `setImage(Image)`——设置接收者将要显示的图像为参数。
- `setMenu(Menu)`——设置接收者的下拉菜单为参数。
- `setSelection(boolean)`——设置接收者的选中状态。
- `setText(String)`——设置接收者的文本。

有用的菜单项创建样式包括:

- SWT.CHECK——创建一个控制开和关的单选框菜单。
- SWT.CASCADE——创建一个带有子菜单的级联菜单。
- SWT.PUSH——创建一个标准菜单项。
- SWT.RADIO——创建一个单选按钮菜单。
- SWT.SEPARATOR——创建一个菜单项分割线。

下列示例创建了一个菜单栏, 具有一个包含两个菜单项和一个分割线的单一菜单 (图4-14)。

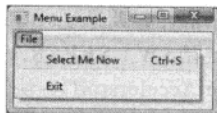


图4-14 菜单示例

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class MenuExample {
    public static void main(String[] args) {
        Display display = new Display();
        final Shell shell = new Shell(display);
        shell.setText("Menu Example");
        shell.setBounds(100, 100, 200, 100);
        Menu bar = new Menu(shell, SWT.BAR);
        shell.setMenuBar(bar);
        MenuItem fileMenu = new MenuItem(bar, SWT.CASCADE);
        fileMenu.setText("&File");
        Menu subMenu = new Menu(shell, SWT.DROP_DOWN);
        fileMenu.setMenu(subMenu);
        MenuItem selectItem = new MenuItem(
            subMenu, SWT.NULL);
        selectItem.setText("&Select Me Now \tCtrl+S");
        selectItem.setAccelerator(SWT.CTRL + 'S');
        selectItem.addSelectionListener(
            new SelectionAdapter(){
                public void widgetSelected(SelectionEvent event){
                    System.out.println("I was selected!");
                }
            });
        MenuItem sep = new MenuItem(subMenu, SWT.SEPARATOR);
        MenuItem exitItem = new MenuItem(subMenu, SWT.NULL);
        exitItem.setText("&Exit");
        exitItem.addSelectionListener(new SelectionAdapter(){
            public void widgetSelected(SelectionEvent event){
                shell.dispose();
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

菜单窗口小部件作为shell的子部件创建。通过setMenuBar()方法将该菜单设置为该shell的菜单栏。然后，一个层叠菜单项被作为File菜单的父部件创建。然后，创建了一个下拉菜单作为shell的子部件，并通过setMenu()方法与File菜单相关联。然后，为该下拉菜单创建了三个菜单项（第二个使用SWT.SEPARATOR创建样式作为分割线）。使用setText()方法设置菜单项的文本，使用setAccelerator()方法设置加速器。为了给菜单项添加行为，添加了一个选择监听器，在该监听器中创建了一个覆盖widgetSelected()方法的SelectionAdapter。

4.2.8 其他窗口小部件

除了以上提到的之外，SWT还提供了许多其他的窗口小部件。SWT支持自定义、模拟的窗口小部件，还支持范围广泛的本地小部件。

有许多附加的原生窗口小部件可用。工具栏（ToolBar）和增强工具栏（CoolBar）通过包含按钮和其他部件的工具栏。扩展栏（ExpandBar）实现了一个可扩展和可折叠的类似抽屉的部件。滚动条、缩放栏和滑动条提供从一个范围中选择一个数字的便利方法。进度条显示了任务的增长进度。日期时间部件（DateTime）允许用户选择一个日期。浏览器提供显示网页的HTML内容方法。链接实现了链接至其他窗口或对话框的简单链接；画布提供了使用绘制任意图像或实现自定义部件的一块空白绘画区域。分割窗提供了一种容器。在该容器中每一个子部件通过一个可移动的分割栏与其他子部件隔开。

同样，还可使用许多自定义的模拟部件。CLabel提供了文本、图像、不同的边框样式和渐变色彩填充的功能；CCombo是标准复合部件的更灵活版本；CTabFolder用于实现Eclipse接口的选项卡元素，并支持不同的样式和渐变色彩填充；而StylesTexts显示完全样式化（粗体、斜体、下划线的、彩色的等）的文本行。

Eclipse Nebula项目（www.eclipse.org/nebula）是自定义SWT部件和其他UI组件的附加源项目。当前可用的部件包括Grid、CDateTime、CTableTree、CompositeTable、PGroup、PShelf、Gallery、FormattedText、DateChooser、CollapsibleButtons、CalendarCombo和GanttChart。

4.3 布局管理

在上一小节展示的每一个示例中，窗口小部件的布局都是十分简单的。部件可以通过两种方法放置。一种是通过setBounds()方法（null布局）相对于它们的父部件放置。另一种是通过使用一个FillLayout将它们的父部件完全填满。Eclipse提供了几个更强大的布局管理算法可以用于根据不同条件美观地放置部件。

Eclipse中的绝大部分布局管理器起源于VisualAge for Smalltalk。尤其是VisualAge for Java中的用来创建向导和对话框的布局管理器。本身它们在转换为Java成为Eclipse框架的一部分之前就是经过良好设计和完整测试的。足够有趣的是，最新的Eclipse布局管理器，FormLayout，是基于VisualAge for Smalltalk中最古老的、最强大的布局管理器。

4.3.1 填充布局（FillLayout）

如同你所见的那样，FillLayout为窗口小部件（如列表或表）提供了一种完全填充其父部件（图4-5或图4-6）的方法。FillLayout可以完成更多的工作，然而因为它提供了一种方法（该方法可以将一组窗口小部件放置为一行或一列），这样每一个窗口小部件都和组中其他窗口小部件大小保持一致（图4-8）。

每一个部件的宽度和高度与该组部件中最大的宽度和高度保持一致。而且，没有选项可以用于控制部件的间隔、页边距或环绕。FillLayout仅仅定义了以下这一个重要属性：

- type——定义了布局的朝向。合法的值为SWT.HORIZONTAL（默认）和SWT.VERTICAL。

与在简单工具栏中的一样，FillLayout对于创建统一的窗口小部件行或列是十分理想的。以下示例创建了一个同样大小的按钮行（图4-15）。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class FillLayoutExample {
    public static void main(String[] args) {
        Button button;
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("FillLayout Example");
        shell.setBounds(100, 100, 400, 75);
        shell.setLayout(new FillLayout());
        for (int i = 1; i <= 8; i++) {
            button = new Button(shell, SWT.PUSH);
            button.setText("B"+i);
            button.addSelectionListener(
                new SelectionAdapter() {
                    public void widgetSelected(
                        SelectionEvent event) {
                        System.out.println(
                            ((Button)event.widget).getText()+
                            "was clicked!");
                    }
                }
            );
            shell.open();
            while (!shell.isDisposed()) {
                if (!display.readAndDispatch()) display.sleep();
            }
            display.dispose();
        }
    }
}
```

默认的，FillLayout是水平朝向的。当按钮被添加至shell时，它们以统一的宽度和高度从左至右一字排开。



图4-15 FillLayout示例

4.3.2 行布局 (RowLayout)

RowLayout与FillLayout十分相似。它也是将窗口小部件以一行或一列的形式放置，但它有许多附加选项可以控制布局。部件间的空白和部件与父容器间的边缘都可以被控制。部件可以被包装为多行、多列或同样大小。RowLayout定义了几个重要属性：

- justify——指定一行中的控件是否完全对齐，控件间是否有额外的空白区域。
- marginBottom——指定布局底部的竖直边缘区域的像素数。默认值是3。

- `marginLeft`——指定布局左侧的水平边缘区域的像素数。默认值是3。
- `marginRight`——指定布局右侧的水平边缘区域的像素数。默认值是3。
- `marginTop`——指定布局顶部的竖直边缘区域的像素数。默认值是3。
- `pack`——指定布局内的所有控件是否以首选大小呈现。如果`pack`是`false`，则所有的控件将具有同样大小。该大小用于调整以适应布局中所有控件的最大的首选项高度和宽度值。
- `Spacing`——指定一个单元格的边界与它的相邻单元格边界之间的像素数。默认值是3。
- `type`——指定布局的朝向。合法的值有`SWT.HORIZONTAL`（默认）和`SWT.VERTICAL`。
- `wrap`——指定如果当前行空间不够时，控件是否在下一行显示。

布局中每一个窗口小部件的宽度和高度可以通过一个`RowData`对象控制。它可以通过`setLayoutData()`方法分配给窗口小部件。`RowData`对象有两个重要属性：

- `width`——以像素形式定义单元的宽度。
- `height`——以像素形式定义单元的高度。

下列示例创建了一个行布局。该行布局具有从窗口帧边缘开始插入的20个平均分配的按钮。决定于父`shell`的大小和形状，该行按钮被包装成一行或多行（图4-16）。

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class RowLayoutExample {
    public static void main(String[] args) {
        Button button;
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("RowLayout Example");
        shell.setBounds(100, 100, 400, 100);
        RowLayout layout = new RowLayout();
        layout.marginLeft = 10;
        layout.marginRight = 10;
        layout.marginTop = 10;
        layout.marginBottom = 10;
        layout.spacing = 10;
        shell.setLayout(layout);
        for (int i = 1; i <= 20; i++) {
            button = new Button(shell, SWT.PUSH);
            button.setText("B" + i);
            button.addSelectionListener(
                new SelectionAdapter() {
                    public void widgetSelected(
                        SelectionEvent event) {
                        System.out.println(
                            ((Button)event.widget).getText() +
                            " was clicked!");
                    }
                });
        }
        shell.open();
    }
}
```



```
while (!shell.isDisposed()) {  
    if (!display.readAndDispatch()) display.sleep();  
}  
display.dispose();  
}  
}
```

默认地, RowLayout是水平排列的。按钮和父shell间的边缘空白通过四个边缘属性设置: marginLeft、marginRight、marginTop和marginBottom。部件间的距离通过使用spacing属性设置。当所有属性被设好之后, 布局通过使用setLayout()分配给该shell。

4.3.3 网格布局 (GridLayout)

绝大部分对话框、向导和首选项页都使用GridLayout布局。它是Eclipse中最常用的, 也是最复杂的布局类之一。

GridLayout以一种具有高可配置性的行与列的网格放置它的子部件。这些行与列提供了许多控制每一个子部件的大小调整行为的选项。

GridLayout定义了以下重要属性。

- horizontalSpacing——定义了一个单元的右边界和相邻单元左边界之间距离的像素数。默认值是5。
- makeColumnsEqualWidth——定义是否所有的列都强制为同样宽度。默认值是false。
- marginWidth——定义网格的左右边界外的边缘所占用的像素数。默认值是5。
- marginHeight——定义网格的上下边界外的边缘所占用的像素数。默认值是5。
- numColumns——定义组成网格的列的数目。默认值是1。
- verticalSpacing——定义一个单元的下边界和相邻单元上边界之间距离的像素数。默认值是5。

布局中的每一个窗口小部件的布局属性可以通过使用GridData对象控制, 该对象可以通过setLayoutData()方法分配给窗口小部件。GridData对象具有以下主要属性:

- grabExcessHorizontalSpace——定义单元格是否应变大以占据网格中额外的水平空间。当网格中的单元格大小由基于窗口小部件和它们的网格数据计算出之后, 复合部件中剩下的额外空间将被分配给那些与额外空间相邻的单元格。
- grabExcessVerticalSpace——指定单元格是否应变大以占据网格中额外的竖直空间。
- heightHint——为窗口小部件指定最小高度 (即为包含该控件的行定义)。
- horizontalAlignment——指定单元格中的窗口小部件的水平对齐方式。其值有SWT.BEGINNING、SWT.CENTER、SWT.END和SWT.FILL。SWT.FILL表示部件将会被改变大小以消费它的网格单元格的整个宽度。
- horizontalIndent——指定窗口小部件与它所在网格单元格的左边界之间的像素数。默认值是0。
- horizontalSpan——指定窗口小部件将要扩展的网格中的列数。默认地, 一个窗口小部件占据网格中的一个单元格。它可以通过增加该值以添加额外的水平单元格。默认值是1。
- verticalAlignment——指定单元格中的窗口小部件的竖直对齐方式。合法的值有SWT.BEGINNING、SWT.CENTER、SWT.END和SWT.FILL。SWT.FILL表示窗口小部件将会

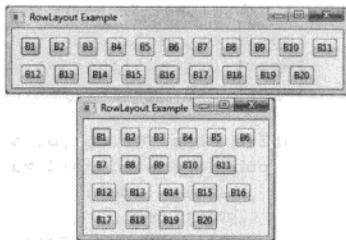


图4-16 RowLayout示例

被调整大小以占据它所在网格单元的整个高度。

- **verticalSpan**——指定窗口小部件将扩展占据的网格的列数。默认地，一个窗口小部件占据网格中的一个单元格。它可以增加该值以添加额外的竖直单元格。默认值是1。
- **widthHint**——指定窗口小部件的最小宽度（即为包含它的列定义）。

以下示例代码创建了一个两列网格布局，包含一个两列扩展标签和两组标签和字段（图4-17）。

```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class GridLayoutExample {
    public static void main(String[] args) {
        Label label;
        Text text;
        GridData gridData;
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("GridLayout Example");
        shell.setBounds(100, 100, 200, 100);
        GridLayout layout = new GridLayout();
        layout.numColumns = 2;
        shell.setLayout(layout);

        label = new Label(shell, SWT.LEFT);
        label.setText("Enter your first and last name");
        gridData = new GridData();
        gridData.horizontalSpan = 2;
        label.setLayoutData(gridData);

        label = new Label(shell, SWT.LEFT);
        label.setText("First:");
        text = new Text(shell, SWT.SINGLE | SWT.BORDER);
        gridData = new GridData();
        gridData.horizontalAlignment = GridData.FILL;
        gridData.grabExcessHorizontalSpace = true;
        text.setLayoutData(gridData);

        label = new Label(shell, SWT.LEFT);
        label.setText("Last:");
        text = new Text(shell, SWT.SINGLE | SWT.BORDER);
        gridData = new GridData();
        gridData.horizontalAlignment = GridData.FILL;
        gridData.grabExcessHorizontalSpace = true;
        text.setLayoutData(gridData);

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

numColumn属性定义了该GridLayout应具有两列。为第一个标签创建的GridData对象的horizontalSpan属性指定了它应扩展为两列。创建的GridData对象具有horizontalAlignment属性,该属性指定每个文本字段应填充整个单元格;grabExcessHorizontalSpace属性指定每一个字段将占据任何剩下的空间。

4.3.4 表单布局 (FormLayout)

没有地方比FormLayout类更能说明Eclipse来源于VisualAge for Smalltalk这一事实了。FormLayout类实现了一种基于附件的布局管理器。FormLayout是最强大的Eclipse布局管理器。它几乎是二十多年前VisualAge for Smalltalk初次使用的布局管理系统的复制品。

通过使用基于附件的布局,你可以独立控制部件的四个边框的改变大小行为。顶部、底部、左侧和右侧可以独立地与父容器或相同容器内的使用固定或相对便宜的其他部件的边界相关联。这项功能被证明是惊人强大的,也可以用于模拟绝大部分其他的布局管理器。

FormLayout类是十分简单的,并且仅定义了容器的边缘。真正的威力在于FormData对象,该对象具有四个不同的FormAttachment对象(每一个对应于一个边框)。FormLayout定义了两个主要属性:

- marginWidth——定义将会被沿着布局的左右边界被放置的水平边缘的像素数。
- marginHeight——定义将会被沿着布局的上下边界被放置的垂直边缘的像素数。

FormData定义几个主要属性:

- top——定义控件上边框的附属属性。
- bottom——定义控件下边框的附属属性。
- left——定义控件的左边框的附属属性。
- right——定义空间的右边框的附属属性。
- width——以像素定义表单中控件的首选宽度。
- height——以像素定义表单中控件的首选高度。

FormAttachment定义了几个主要属性:

- alignment——指定控件相关联的控件边框的对齐方式。SWT.DEFAULT表示部件将会与指定空间的邻近边相关联。对于顶部和底部附属属性,SWT.TOP、SWT.BOTTOM和SWT.CENTER用于表示部件的指定边与控件的指定边的附属关系。对于左侧和右侧附属属性,SWT.LEFT、SWT.RIGHT和SWT.CENTER用于表示部件的指定边与控件的指定边的附属关系。比如,使用SWT.TOP表示附属部件的上边框将与指定控件的上边框相关联。
- control——指定附属部件将要附属的目标控件。
- denominator——指定用于定义附件的等式 $y = ax + b$ 中的“a”的分母。
- numerator——指定用于定义附件的等式 $y = ax + b$ 中的“a”的分子。
- offset——以像素指定控件边与附件位置之间的偏移量,可以是正值,也可以是负值。这是定义附件的等式 $y = ax + b$ 中的“b”。

以下示例创建了一个简单的表单布局。在该布局的右下角有两个按钮,一个文本框填充剩下空间(参见图4-18以了解紧邻运行窗口的两个示例的不同大小的窗口)。Cancel按钮与右下角相关联。OK按钮与窗口的底部、Cancel按钮的左边界相关联。文本框与窗口的上边框、左边框和右边框,以

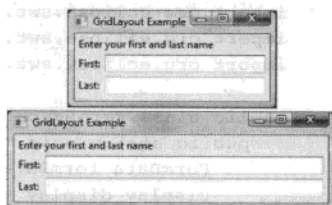


图4-17 网格布局示例

及Cancel按钮的上边框相关联。

```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class FormLayoutExample {
    public static void main(String[] args) {
        FormData formData;
        Display display = new Display();
        final Shell shell = new Shell(display);
        shell.setText("FormLayout Example");
        shell.setBounds(100, 100, 220, 180);
        shell.setLayout(new FormLayout());

        Button cancelButton = new Button(shell, SWT.PUSH);
        cancelButton.setText("Cancel");
        formData = new FormData();
        formData.right = new FormAttachment(100, -5);
        formData.bottom = new FormAttachment(100, -5);
        cancelButton.setLayoutData(formData);

        Button okButton = new Button(shell, SWT.PUSH);
        okButton.setText("OK");
        formData = new FormData();
        formData.right = new FormAttachment(cancelButton, -5);
        formData.bottom = new FormAttachment(100, -5);
        okButton.setLayoutData(formData);

        Text text = new Text(shell, SWT.MULTI | SWT.BORDER);
        formData = new FormData();
        formData.top = new FormAttachment(0, 5);
        formData.bottom = new FormAttachment(
            cancelButton, -5);
        formData.left = new FormAttachment(0, 5);
        formData.right = new FormAttachment(100, -5);
        text.setLayoutData(formData);

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

分配给Cancel按钮的FormData具有一个right和bottom附属属性。该附属属性与shell的右下角相关联。每一个FormAttachment对象的第一个参数是用于初始关联的shell的百分比（以0%的值首先左上角）。值100指定了右下边缘，与左上边缘相对。

第二个参数代表从附加点开始的固定偏移（正值指向右侧和下面）。值-5表示部件应从底部和右侧偏移5个像素。

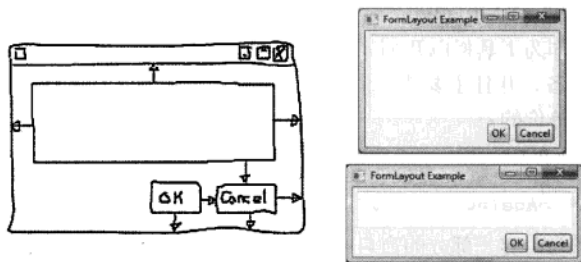


图4-18 FormLayout示例

请注意左侧和顶部附件没有被指定。不指定它们的值将使部件采用它的首选宽度和高度。

OK按钮也附加于shell的底部。它的右边框被附加于Cancel按钮的左边界，而不是shell本身。这为OK按钮提供了将它自身相对于Cancel按钮的首选大小来放置的方法。这种样式对于在设计时不知道按钮的文本的放置位置的国际化应用程序（因此也对于它们的首选大小）特别有用。

最后，文本框被关联为对于shell的左侧，右侧和顶部边框固定的5像素的偏移。文本框的底部与Cancel按钮的顶部的5像素偏移相关联。

4.4 资源管理

与SWT其他部分的设计一致，颜色、字体和图像也是对它们的系统对应项进行了一定的包装。而这些封装平台对应部件的包装部件在不再需要时，必须明确销毁。

基本原则是，如果你从某个其他地方访问颜色、字体或图像，你不需要为其担心。另一方面，如果你创建了该资源，那么就必须在不需要的时候销毁它。对于在你的程序中需要经常访问到的资源，可以考虑创建一个资源管理器来管理它们并在你程序关闭时销毁所有资源。

4.4.1 颜色

颜色是为特定设备（该设备可以是null，表示默认设备）创建的，由三个整型值表示每一个颜色元素（红色、绿色和蓝色），从0到255（比如，new color(null, 255, 0, 0)创建了红色）。部件的前景和背景颜色可以分别通过setForeground()和setBackground()方法设置。

要使用由系统预定义的颜色，如窗口背景颜色或按钮背景颜色，你可以使用Display.getSystemColor(int)方法，该方法接受目标颜色的标识符作为参数。你不需要销毁所有通过这种方式获得的颜色。

4.4.2 字体

与颜色类似，字体也是为特定设备创建的，并由字体名称（如Arial、Times等）、高度和样式（SWT.NORMAL, SWT.BOLD或SWT.ITALIC的组合）所描述。字体可以由指定名称、高度和样式直接创建，也可以引用一个包含这三个值的FontData对象创建。比如，new Font(null, "Arial", 10, SWT.BOLD)创建了一个10像素的粗体Arial字体。部件字体可以通过使用setFont()方法设置。

4.4.3 图像

图像在工具栏、按钮、标签、树和表中经常被用到。Eclipse支持载入和保存图像于多种常用格

式,如GIF、JPEG、PNG、BMP (Windows位图)和ICO (Windows图标)。一些格式,如GIF,支持透明,这让它们对于作为工具栏以及列表和表中的项目装饰是理想的。

图像创建于特定设备,并且主要从一个指定文件载入或创建于一个设备无关的ImageData对象。比如,以下两种方法是等价的:

```
Image img = new Image(null, "c:\\my_button.gif")
ImageData data = new ImageData("c:\\my_button.gif");
Image img = new Image(null, data);
```

对于支持将图像作为内容一部分的窗口小部件(比如标签和按钮)来说,可以使用setImage()方法设置部件的图像。为了了解关于图像缓存和ImageDescriptor的信息,参见7.7节。

4.5 GUI构建器 (GUI Builder)

手工创建复杂的用户界面是一项十分具有挑战性的工作。这对于SWT和Swing都是如此。图形用户界面 (Graphical User Interface, GUI) 创建程序可以显著降低用于创建大部分Eclipse插件所需的用户界面元素 (视图、编辑器、透视图、首选项页等) 的时间。

利用GUI构建器,你可以十分快速地创建复杂窗口。它也为你的生成相应的Java代码。你可以通过拖放轻松地添加窗口小部件,为你的窗口小部件添加事件处理器,使用属性编辑器编辑部件的不同属性,国际化你的程序等。

对于Eclipse插件开发真正有用的是,GUI创建程序需要支持由SWT、JFace和RCP支持的所有小部件、布局管理器和用户界面颜色。此外,一个GUI创建程序还应该是双向的,良好重构的,并具有反向工程手写Java GUI代码的功能。

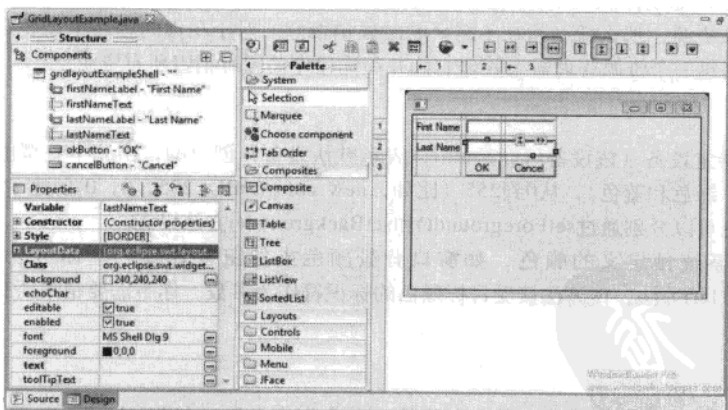


图4-19 WindowBuilder Pro

虽然有很多Eclipse GUI创建程序,但唯一满足所有这些条件的是Instantiations公司的WindowBuilder Pro (图4-19)。WindowBuilder是一个Eclipse商业插件(和许多基于Eclipse的IDE一样,如MyEclipse和JBuilder)。它对于工作于开源项目的开发者可以免费获取。它创建一个抽象语法树 (Abstract Syntax Tree, AST) 来浏览源代码,并使用GEF来显示和管理可视层。

WindowBuilder可以读取并写入几乎所有的格式，并支持任意代码编辑（在任意位置做出修改，而不是仅仅在特定区域）和大部分用户重构（移动、重命名和准确细分方法）。

4.6 总结

SWT是一个良好设计的原生UI Java库。它来源于由IBM和OTI多年以来的类似工作经验。它是Eclipse本身的原生UI库，并会被任何你创建的Eclipse插件频繁使用。SWT也可以满足创建独立运行的Java程序的需要，这样就不需要任何其他Eclipse框架了。

SWT包括一个丰富的内建部件集。只要存在对应部件，这些内建部件被映射到系统原生部件。如果特定平台不存在对应部件，它将会生成模拟部件。SWT还包括了一个广泛的布局管理类的数组。从简单的FillLayout到最复杂的GridLayout和FormLayout。通过这些部件和布局管理器，你可以创建插件所需的任何用户界面。

参考文献

本书资源 (2.9节).

Eclipse SWT (<http://www.eclipse.org/swt>)

and Eclipse SWT snippets (<http://www.eclipse.org/swt/snippets>)

Northover, Steve, and Mike Wilson, *SWT: The Standard Widget Toolkit*. Addison-Wesley, Boston, 2004.

Harris, Robert, and Rob Warner, *The Definitive Guide to SWT and JFace*. Apress, Berkeley, CA, 2004.

Holder, Stephen, Stanford Ng, and Laurent Mihalkovic, *SWT/JFace in Action: GUI Design with Eclipse 3.0*. Manning Publications, Greenwich, CT, 2004.

Cornu, Christophe, "A Small Cup of SWT," IBM OTI Labs, September 19, 2003 (www.eclipse.org/articles/Article-small-cup-of-swt/pocket-PC.html).

SWT Graph (<http://swtgraph.sourceforge.net/examples.php>)

Winchester, Joe, "Taking a Look at SWT Images," IBM, September 10, 2003 (www.eclipse.org/articles/Article-SWT-images/graphics-resources.html).

Irvine, Veronika, "Drag and Drop—Adding Drag and Drop to an SWT Application," IBM, August 25, 2003 (www.eclipse.org/articles/Article-SWTDND/DND-in-SWT.html).

Arthorne, John, "Drag and Drop in the Eclipse UI," IBM, August 25, 2003 (www.eclipse.org/articles/Article-Workbench-DND/drag_drop.html).

Bordeau, Eric, "Using Native Drag and Drop with GEF," IBM, August 25, 2003 (www.eclipse.org/articles/Article-GEF-dnd/GEF-dnd.html).

Savarese, Daniel F., "Eclipse vs. Swing," JavaPro, December 2002 (www.ftponline.com/javapro/2002_12/magazine/columns/proshop/default_pf.aspx).

Majewski, Bo, "Using OpenGL with SWT," Cisco Systems, Inc., April 15, 2005 (www.eclipse.org/articles/Article-SWT-OpenGL/opengl.html).

Kues, Lynne, and Knut Radloff, "Getting Your Feet Wet with the SWT Styled- Text Widget," OTI, July 19, 2004 (www.eclipse.org/articles/StyledText%201/article1.html).

Kues, Lynne, and Knut Radloff, "Into the Deep End of the SWT StyledText Widget," OTI, September 18, 2002 (www.eclipse.org/articles/Styled-Text%20article2.html).

Li, Chengdong, "A Basic Image Viewer," University of Kentucky, March 15, 2004 (www.eclipse.org/articles/Article-Image-Viewer/Image_viewer.html).

MacLeod, Carolyn, and Shantha Ramachandran, "Understanding Layouts in SWT," OTI, May 2, 2002 (www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm).

Northover, Steve, "SWT: The Standard Widget Toolkit—PART 1: Implementation Strategy for Java™ Natives," OTI, March 22, 2001 (www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html).

MacLeod, Carolyn, and Steve Northover, "SWT: The Standard Widget Toolkit—PART 2: Managing Operating System Resources," OTI, November 27, 2001 (www.eclipse.org/articles/swt-design-2/swt-design-2.html).

Moody, James, and Carolyn MacLeod, "SWT Color Model," OTI, April 24, 2001 (www.eclipse.org/articles/Article-SWT-Color-Model/swt-colormodel.htm).

Irvine, Veronika, "ActiveX Support In SWT: How Do I Include an OLE Document or ActiveX Control in My Eclipse Plug-in?," OTI, March 22, 2001 (www.eclipse.org/articles/Article-ActiveX%20Support%20in%20SWT/ActiveX%20Support%20in%20SWT.html).



第5章 JFace查看器

虽然SWT提供了原生系统窗口小部件的直接接口，但它被局限于只能使用简单数据类型，主要有字符串、数字和图像。这对于许多程序是没有问题的，但处理列表、表、树和文本部件中需要用的面向对象的数据时，SWT将是严重不匹配的。这是JFace查看器逐步为SWT部件提供面向对象包装器的原因。

5.1 面向列表的查看器

JFace列表查看器，如ListViewer、TableView和TreeView，允许你直接使用域模型对象（如公司、个人、部门等事务对象）而不需要将它们分解为基本的字符串、数字和图像元素。查看器通过在获取项目标签（包括图像和文本）后为这些对象提供适配器接口来完成这项工作。查看器可以用于访问项目的后代（在树中）、从列表中选择个项目、在列表中排列项目、在列表中过滤项目和将任意输入转换为适用于底层SWT部件的列表（图5-1）。

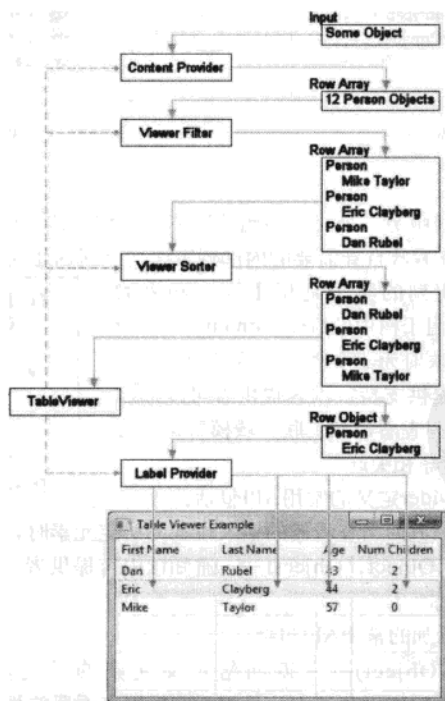


图5-1 查看器与适配器的关系

5.1.1 标签提供者

标签提供者是列表查看器最经常使用到的适配器类型之一。该适配器用于将域模型对象映射为可在查看器的部件中显示的一个或多个图像和文本字符串。

标签提供者的两种最常用类型是用于列表和树中的ILabelProvider（图5-2）和用于表中的ITableLabelProvider（图5-3）。前者将项目映射为一个单一图像和文本标签，后者将项目映射为多个图像和文本标签（每一组对应于表中的一列）。标签提供者通过使用setLabelProvider()与查看器相关联。

由ILabelProvider定义的常用API包括：

- getImage(Object)——返回给定元素的标签的图像。
- getText(Object)——返回给定元素的标签的文本。
- 由ITableLabelProvider定义的常用API包括：
- getColumnImage(Object, int)——返回给定元素的给定列的标签图像。
- getColumnText(Object, int)——返回给定元素的给定列的标签文本。

为了了解标签提供者的示例，参见5.1.6节。

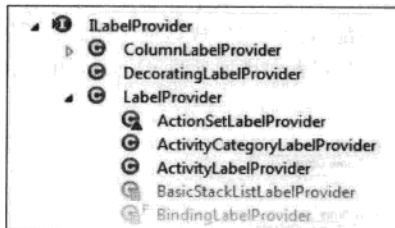


图5-2 LabelProvider层次结构

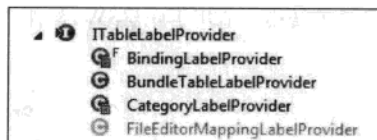


图5-3 TableLabelProvider层次结构

5.1.2 内容提供者

内容提供者是列表查看器中另一个经常用到的适配器类型。该提供者用于在作为查看器的输入的一个或多个域对象模型和查看器自身需要的内部列表结构之间创建映射。

内容提供者两个最常用到的类型是用于列表和表的IStructuredContentProvider和用于树中的ITreeContentProvider（图5-4）。前者将域对象模型映射为一个数组，后者为在树中获取项目的父项目或子项目提供支持。内容提供者可以通过使用setContentProvider()方法与查看器相关联。域模型输入可通过使用setInput()方法与查看器相关联。

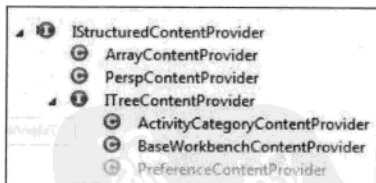


图5-4 ContentProvider层次结构

由IStructuredContentProvider定义的常用API包括：

- getElements(Object)——返回当查看器的输入设置为给定元素时，查看器中显示的元素。
- inputChanged(Viewer, Object, Object)——通知该内容提供者，给定查看器的输入已经被重定向至一个不同的元素。

由ITreeContentProvider添加的常用API包括：

- Object[] getChildren(Object)——返回给定父元素的子元素。该方法和前面介绍的getElements(Object)方法的区别是该方法是用于获取树查看器的根元素，而getChildren(Object)用于获取树中给定父元素（包括根）的后代。

- getParent(Object)——返回给定元素的父元素或null，表示该元素的父元素不可由计算获得。
- hasChildren(Object)——返回给定元素是否具有后代。

为了了解内容提供者的示例，参见5.1.6节。

5.1.3 查看器排序器

查看器排序器 (viewer sorter) (参见图5-5以了解ViewerSorter层次结构) 用于将由内容提供者 (图5-1) 提供的元素进行排序。如果查看器不具有排序器，元素将以由内容提供者返回的顺序显示。查看器排序器可以通过使用 setSorter() 与查看器相关联。

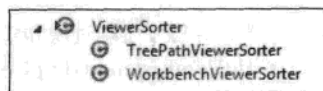


图5-5 ViewerSorter层次结构

默认排序算法使用两步过程。第一步，它将元素进行分组 (从0到n标记)；第二步，它根据由标签提供者返回的文本标签对每一组进行排序。默认地，所有的项目都在一个组中，因此所有的项目都根据它们的文本标签进行排序。你的程序可以覆盖默认的分组策略和默认的比较方法，使用其他条件而不是项目的文本标签。

由ViewerSorter定义的常用API包括：

- category(Object)——返回给定元素的组。
- compare(Viewer, Object, Object)——根据第一个元素是小于等于或大于第二个元素返回一个负值、0或正值。
- getCollator()——返回用于排序元素的整理器。
- isSorterProperty(Object, String)——返回该查看器排序器是否应受给定元素的给定属性的变化的影响。
- sort(Viewer viewer, Object[])——对给定的元素进行排序以修改给定的数组。

为了了解查看器排序器的示例，参见5.1.6节。

5.1.4 查看器过滤器

查看器过滤器 (viewer filter) (参见图5-6以了解ViewerFilter层次结构) 用于显示由内容提供者 (图5-1) 提供的元素的子集。如果一个视图不具有查看器过滤器，所有的元素都将被显示。查看器过滤器可以通过使用 setFilter() 方法与查看器关联。

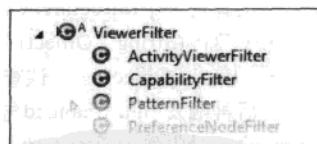


图5-6 ViewerFilter层次结构

由ViewerFilter定义的常用API在下面列出。简单的查看器过滤器仅需覆盖 select(Viewer, Object, Object) 方法以决定一个对象是否应在查看器中显示。

- filter(Viewer, Object, Object[])——过滤给定查看器的给定元素。该方法的默认实现调用下面 select(Viewer, Object, Object) 方法。
- isFilterProperty(Object, String)——返回该查看器过滤器是否由给定元素的给定属性的变化所影响。该方法的默认实现返回 false。
- select(Viewer, Object, Object)——返回给定元素是否可以通过该过滤器。

5.1.5 StructuredViewer类

StructuredViewer类是列表查看器、表查看器和树查看器的抽象超类 (图5-7)。它定义了大量对于每一个类都相同的常用API。

- `addDoubleClickListener(IDoubleClickListener)`——在该查看器中为双击添加监听器。
- `addDragSupport(int, Transfer[], DragSourceListener)`——为通过用户拖放操作将项目拖曳出该查看器提供支持。
- `addDropSupport(int, Transfer[], DropTargetListener)`——为通过用户拖放操作将项目拖入该查看器提供支持。
- `addFilter(ViewerFilter)`——为该查看器添加给定过滤器，并触发重新过滤和重新排列这些元素。
- `addHelpListener(HelpListener)`——为该查看器的帮助请求添加监听器。
- `addOpenListener(IOpenListener)`——为该查看器的选择打开添加监听器。
- `addSelectionChangedListener(ISelectionChangedListener)`——为该选择提供者的选择更改添加监听器。
- `addPostSelectionChangedListener(ISelectionChangedListener)`——为该查看器中的选择后操作添加监听器。
- `getSelection()`——该方法的StructuredViewer实现返回结果为IStructuredSelection。
- `refresh()`——使用从该查看器的模型新获取的信息完全刷新该查看器。
- `refresh(boolean)`——使用从该查看器的模型新获取的信息刷新该查看器。
- `refresh(Object)`——从给定的元素开始刷新该查看器。
- `refresh(Object, boolean)`——从给定的元素开始刷新该查看器。
- `resetFilters()`——绕过该查看器的过滤器，触发对元素的重新过滤和重新排列。
- `setComparer(IElementComparer)`——设置比较器以用于比较元素，或设为null以使用元素自身的默认equals和hashCode方法。
- `setContentProvider(IContentProvider)`——该方法的StructuredViewer实现，验证以保证内容提供者是一个IStructuredContentProvider。
- `setData(String, Object)`——将给定名称的属性设为给定值，如果要移除属性则设为null。
- `setInput(Object)`——该查看器方法的ContentViewer实现在内容提供者上触发inputChanged，然后再触发inputChanged句柄方法。内容提供者的getElements(Object)方法稍后被调用，使用该输入对象作为它的参数，以确定查看器的根级元素。
- `setSelection(ISelection, boolean)`——该方法的StructuredViewer实现更新基于指定选择的当前查看器的选择。
- `setSorter(ViewerSorter)`——设置该查看器的排序器和触发器，以用于重新过滤和重新排列该查看器的元素。
- `setUseHashlookup(boolean)`——设置该结构化的查看器是否使用一个内部hash表以加速元素和SWT项目间的映射。
- `update(Object[], String[])`——当给定元素一个或多个属性改变时更新它的表示。
- `update(Object, String[])`——当给定元素一个或多个属性改变时更新它的表示。

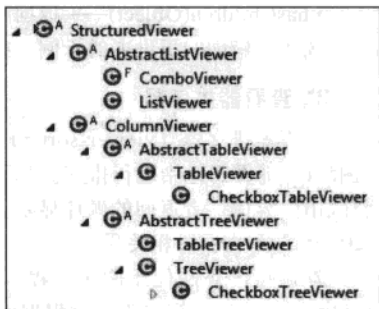


图5-7 StructuredViewer层次结构

5.1.6 ListViewer类

ListViewer类封装List窗口小部件。它用于查看一组对象，而不是一组字符串。列表查看器需为了通过标签和内容提供者进行设置。常用API包括：

- add(Object)——将指定元素添加至该列表查看器。
- add(Object[])——将指定的多个元素添加至该列表查看器。
- getControl()——返回与该查看器关联的主控件。
- getElementAt(int)——从该列表查看器中返回给定索引的元素。
- getList()——返回该列表查看器的列表控件。
- remove(Object)——从该列表查看器移除指定元素。
- remove(Object[])——从该列表查看器移除指定的多个元素。
- reveal(Object)——保证给定元素是可见的。如果需要则将滚动查看器。
- setLabelProvider(IBaseLabelProvider)——该Viewer框架方法的列表查看器实现保证给定的标签提供者是一个ILabelProvider实例。

在以下的几个示例中要使用的Person域模型类与下列代码类似。

```
public class Person {
    public String firstName = "John";
    public String lastName = "Doe";
    public int age = 37;
    public Person[] children = new Person[0];
    public Person parent = null;
    public Person(String firstName, String lastName,
        int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public Person(String firstName, String lastName,
        int age, Person[] children) {
        this(firstName, lastName, age);
        this.children = children;
        for (int i = 0; i < children.length; i++) {
            children[i].parent = this;
        }
    }
    public static Person[] example() {
        return new Person[] {
            new Person("Dan", "Rubel", 41, new Person[] {
                new Person("Beth", "Rubel", 11),
                new Person("David", "Rubel", 6)}),
            new Person("Eric", "Clayberg", 42, new Person[] {
                new Person("Lauren", "Clayberg", 9),
                new Person("Lee", "Clayberg", 7)}),
            new Person("Mike", "Taylor", 55)
        };
    }
    public String toString() {
        return firstName + " " + lastName;
    }
}
```

以下示例代码使用标签提供者、内容提供者和查看器过滤器（图5-8）创建了一个列表查看器。请注意：要单独运行JFace示例，你需要将以下内容添加至Java BuildPath（插件版本号必须与Eclipse中使用的一致）。

```
ECLIPSE_HOME/plugins/org.eclipse.equinox.common_3.X.X.vXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.core.runtime_3.X.X.vXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.core.commands.X.X.vXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.jface_3.X.X.vXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.jface.text_3.X.X.vXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.text_3.X.X.vXXXXXXX.jar
```

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
```

```
public class ListViewExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("List Viewer Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final ListViewer listViewer =
            new ListViewer(shell, SWT.SINGLE);
        listViewer.setLabelProvider(
            new PersonListLabelProvider());
        listViewer.setContentProvider(
            new ArrayContentProvider());
        listViewer.setInput(Person.example());
        listViewer.setSorter(new ViewerSorter(){
            public int compare(
                Viewer viewer, Object p1, Object p2) {
                return ((Person) p1).lastName
                    .compareToIgnoreCase(((Person) p2).lastName);
            }
        });
        listViewer.addSelectionChangedListener(
            new ISelectionChangedListener() {
                public void selectionChanged(
                    SelectionChangedEvent event) {
                    IStructuredSelection selection =
                        (IStructuredSelection) event.getSelection();
                    System.out.println("Selected: " +
                        selection.getFirstElement());
                }
            });
        listViewer.addDoubleClickListener(
            new IDoubleClickListener() {
                public void doubleClick(DoubleClickEvent event) {
                    IStructuredSelection selection =
                        (IStructuredSelection) event.getSelection();
                    System.out.println("Double Clicked: " +
                        selection.getFirstElement());
                }
            });
    }
}
```

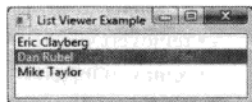


图5-8 ListViewer示例

```
});
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()) display.sleep();
}
display.dispose();
}
}
```

在创建了列表查看器后，标签提供者通过setLabelProvider()方法设置，内容提供者通过setContentProvider()方法设置。标签提供者PersonListLabelProvider返回一个由人员的名和姓组成的文本标签，而且不返回图标。该类与以下代码类似：

```
public class PersonListLabelProvider extends LabelProvider {
    public Image getImage(Object element) {
        return null;
    }
    public String getText(Object element) {
        Person person = (Person) element;
        return person.firstName + " " + person.lastName;
    }
}
```

对于内容提供者，使用内建的ArrayContentProvider类将输入集映射至一个数组。输入对象通过setInput()方法设置。查看器排序器定义了一个自定义的compare()方法。该方法对元素基于个人的姓进行排序。最后，一个selectionChanged监听器和一个doubleClick监听器分别覆盖selectionChanged()方法和doubleClick()方法，并被添加至相应元素。

5.1.7 TableViewer类

TableViewer类封装Table窗口小部件。表查看器提供了一个可编辑、竖直、多列的项目表。它为列表中的每一个项目显示了一行单元格。每一个单元格代表那一行所在的项目的一个不同属性。表查看器需为了通过标签提供者、内容提供者和一个列集设置相关属性。

CheckboxTableViewer通过添加对使独立项目变灰，开关与项目关联的单选框增强了其功能。常用API包括：

- add(Object)——将给定的元素添加至该表查看器。当单个元素已经被添加至模型时，应当调用该方法（由内容提供者调用），以使查看器精确反映该模型。该方法仅仅影响查看器，而不是模型。
- add(Object[])——将给定的元素添加至该表查看器。当多个元素已经被添加至模型时，应当调用该方法（由内容提供者调用），以使查看器精确反映该模型。该方法仅仅影响查看器，而不是模型。
- cancelEditing()——取消一个当前活动的单元格编辑器。
- editElement(Object, int)——开始编辑给定的元素。
- getElementAt(int)——返回该表查看器中给定索引位置的元素。
- getTable()——返回该表查看器的表控件。
- insert(Object, int)——将给定元素添加至该表查看器的给定位置。
- isCellEditorActive()——返回是否存在一个活动单元格编辑器。
- remove(Object)——从该表查看器移除给定的元素。该方法应在单个元素已经从模型中移除时

调用（由内容提供者），以使查看者准备反映模型。该方法仅影响查看器，不影响模型。

- `remove(Object[])`——从该表查看器移除给定的多个元素。该方法应在多个元素已经从模型中移除时调用（由内容提供者），以使查看者准备反映模型。该方法仅影响查看器，不影响模型。
- `reveal(Object)`——保证给定元素是可见的，如果需要则滚动查看器。
- `setCellEditors(CellEditor[])`——设置该表查看器的单元格编辑器。
- `setColumnProperties(String[])`——设置该表查看器的列属性。
- `setLabelProvider(IBaseLabelProvider)`——该Viewer框架方法的表查看器实现保证给定的标签提供者是ITableLabelProvider或ILabelProvider的实例。

CheckboxTableViewer添加了以下常用API：

- `addCheckStateListener(ICheckStateListener)`——为该查看器中元素的选中状态更改添加监听器。
- `getChecked(Object)`——返回给定元素的选中状态。
- `getCheckedElements()`——返回一个对应于该查看器中选中表项目的元素列表。
- `getGrayed(Object)`——返回给定元素的无效状态。
- `getGrayedElements()`——返回一个对应于该查看器中无效节点的元素列表。
- `setAllChecked(boolean)`——将该查看器的所有元素的选中状态设置为给定值。
- `setAllGrayed(boolean)`——将该查看器的所有元素的无效状态设置为给定值。
- `setChecked(Object, boolean)`——设置该查看器中给定元素的选中状态。
- `setCheckedElements(Object[])`——设置该查看器中哪些节点被选中。
- `setGrayed(Object, boolean)`——为该查看器的给定元素设置无效状态。
- `setGrayedElements(Object[])`——设置该查看器中哪些节点是无效的。

以下示例代码创建了一个具有一个标签提供者、一个内容提供者和四列的表查看器（图5-9）。

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TableViewerExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Table Viewer Example");
        shell.setBounds(100, 100, 325, 200);
        shell.setLayout(new FillLayout());

        final TableViewer tableViewer = new TableViewer(
            shell, SWT.SINGLE | SWT.FULL_SELECTION);
        final Table table = tableViewer.getTable();
        table.setHeaderVisible(true);
        table.setLinesVisible(true);

        String[] columnNames = new String[] {
            "First Name", "Last Name", "Age", "Num Children"};
        int[] columnWidths = new int[] {
            100, 100, 35, 75};
        int[] columnAlignments = new int[] {
```

First Name	Last Name	Age	Num Children
Dan	Rubel	43	2
Eric	Clayberg	44	2
Mike	Taylor	57	0

图5-9 TableViewer示例

```

        SWT.LEFT, SWT.LEFT, SWT.CENTER, SWT.CENTER};
    for (int i = 0; i < columnNames.length; i++) {
        TableColumn tableColumn =
            new TableColumn(table, columnAlignments [i]);
        tableColumn.setText(columnNames [i]);
        tableColumn.setWidth(columnWidths [i]);
    }

    tableViewer.setLabelProvider(
        new PersonTableLabelProvider());
    tableViewer.setContentProvider(
        new ArrayContentProvider());
    tableViewer.setInput(Person.example());

    shell.open();
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) display.sleep();
    }
    display.dispose();
}
}

```

在创建该表查看器之后，在表查看器的底层表中，通过调用 `setHeaderVisible()` 和 `setLinesVisible()` 方法将列头部和行设为可见。然后，使用不同的对齐方式将四列添加至表中。头部的文本和每一列的宽度通过 `setText()` 和 `setWidth()` 方法设置（参见 7.8 节）。

标签提供者通过使用 `setLabelProvider()` 方法设置。内容提供者通过使用 `setContentProvider()` 方法设置。标签提供者 `PersonTableLabelProvider`，返回表中每一列的文本标签，不返回图标。该类与以下代码类似：

```

import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.*;

public class PersonTableLabelProvider
    extends LabelProvider
    implements ITableLabelProvider {
    public Image getColumnImage(Object element, int index) {
        return null;
    }
    public String getColumnText(Object element, int index) {
        Person person = (Person) element;
        switch (index) {
            case 0 :
                return person.firstName;
            case 1 :
                return person.lastName;
            case 2 :
                return Integer.toString(person.age);
            case 3 :
                return Integer.toString(person.children.length);
            default :
                return "unknown " + index;
        }
    }
}

```

5.1.8 TreeViewer类

TreeViewer类封装Tree窗口小部件。树查看器显示了父子关系的对象层次列表。该查看器需要使用标签和内容查看器进行设置。CheckboxTreeViewer更进一步增强了其功能。它可以使独立的项目无效，打开或关闭项目相关的单选框。常用API包括：

- add(Object, Object)——将给定的子元素添加至该查看器以作为给定父元素的后代。
- add(Object, Object[])——将给定的多个子元素添加至该查看器以作为给定父元素的后代。
- addTreeListener(ITreeViewerListener)——为该查看器中的展开和收缩事件添加监听器。
- collapseAll()——从根节点开始收缩该查看器树的所有节点。
- collapseToLevel(Object, int)——收缩给定级别的给定元素开始的子树。
- expandAll()——从根节点开始展开该查看器树的所有节点。
- expandToLevel(int)——从给定的级别开始展开查看器树的根。
- expandToLevel(Object, int)——展开给定元素的所有祖先，以使给定元素在查看器树控件中可见，然后扩展给定级别的给定元素开始的子树。
- getExpandedElements()——返回一个对应于该查看器树的已扩展节点的元素列表，包括当前隐藏的被标记为已扩展但在一个收缩的祖先下面的节点。
- getExpandedState(Object)——返回对应给定元素的节点是展开的还是收缩的。
- Tree getTree()——返回该树查看器的树控件。
- getVisibleExpandedElements()——获取对于用户可见的已展开元素。
- isExpandable(Object)——返回表示给定元素的树节点是否可以展开的值。
- remove(Object)——从该查看器移除给定元素。
- remove(Object[])——从该查看器移除给定的多个元素。
- reveal(Object)——保证给定元素是可见的，如果需要则将滚动查看器。
- scrollDown(int, int)——从给定的相对于显示的坐标开始逐个项目向下滚动查看器控件。
- scrollUp(int, int)——从给定的相对于显示的坐标开始逐个项目向上滚动查看器控件。
- setAutoExpandLevel(int)——设置自动展开的级别。
- setContentProvider(IContentProvider)——该方法的AbstractTreeViewer实现，验证以保证内容提供者是一个ITreeContentProvider。
- setExpandedElements(Object[])——设置该查看器树中哪些节点可以展开。
- setExpandedState(Object, boolean)——设置对应于给定元素的节点时展开还是收缩的。
- setLabelProvider(ILabelProvider)——该Viewer框架方法的树查看器实现保证给定的标签提供者是一个ILabelProvider的实例。

CheckboxTreeViewer添加了下列常用API：

- addCheckStateListener(ICheckStateListener)——为该查看器元素的选中状态变更添加监听器。
- getChecked(Object)——返回给定元素的确认状态。
- getCheckedElements()——返回一个在当前查看器树中的选中元素的列表，包含当前隐藏的但被标记为选中的但在一个收缩的祖先下面的节点。
- getGrayed(Object)——返回给定元素的无效状态。
- getGrayedElements()——返回当前查看器树的无效节点的列表，包含当前隐藏的但被标记为选中的，但在一个收缩的祖先下面的节点。

- `setChecked(Object, boolean)`——为当前查看器的给定元素设置选中状态。
- `setCheckedElements(Object[])`——设置当前查看器树的哪些元素是选中的。
- `setGrayChecked(Object, boolean)`——选中并使选项无效，而不是调用`setGrayed`和`setChecked`作为优化。
- `setGrayed(Object, boolean)`——设置当前查看器的给定元素的无效状态。
- `setGrayedElements(Object[])`——设置当前查看器树哪些元素是无效的。
- `setParentsGrayed(Object, boolean)`——在当前查看器中为给定元素和它的父元素设置无效状态。
- `setSubtreeChecked(Object, boolean)`——为当前查看器中的给定元素和它的可见后代设置选中状态。

以下示例创建了一个具有一个标签提供者 and 内容提供者的树查看器 (图5-10)。

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TreeViewerExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Tree Viewer Example");
        shell.setBounds(100, 100, 200, 200);
        shell.setLayout(new FillLayout());

        final TreeViewer treeViewer =
            new TreeViewer(shell, SWT.SINGLE);
        treeViewer.setLabelProvider(
            new PersonListLabelProvider());
        treeViewer.setContentProvider(
            new PersonTreeContentProvider());
        treeViewer.setInput(Person.example());

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

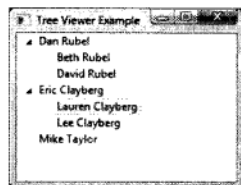


图5-10 TreeViewer示例

在创建了树查看器之后，标签提供者通过`setLabelProvider()`方法设置，内容提供者通过`setContentProvider()`方法设置。该内容提供者`PersonTreeContentProvider`返回每一个项目的父项目和后代。该类与下列代码类似：

```
import org.eclipse.jface.viewers.*;

public class PersonTreeContentProvider
    extends ArrayContentProvider
    implements ITreeContentProvider {
```

```

public Object[] getChildren(Object parentElement) {
    Person person = (Person) parentElement;
    return person.children;
}

public Object getParent(Object element) {
    Person person = (Person) element;
    return person.parent;
}

public boolean hasChildren(Object element) {
    Person person = (Person) element;
    return person.children.length > 0;
}
}

```

5.2 文本查看器

TextViewer类封装StyledText部件（参见图5-11以了解TextViewer层次结构）。文本的单个运行可能具有与它们相关联的不同的样式，包括前景颜色、背景颜色和粗体。文查看器为客户端提供了一个文本模型，并管理用于文本部件的文档中的样式文本信息的转换。

常用API包括：

- addTextListener(ITextListener)——为当前查看器添加一个文本监听器。
- appendVerifyKeyListener(VerifyKeyListener)——将一个验证关键字的监听器附加至该查看器的验证关键字的监听器列表末尾。
- canDoOperation(int)——返回由给定的操作代码指定的操作是否能被执行。
- changeTextPresentation(TextPresentation, boolean)——应用编码于给定文本表达中的颜色信息。
- doOperation(int)——执行由目标的操作代码指定的操作。
- enableOperation(int, boolean)——使给定的文本操作有效或无效。
- getSelectedRange()——返回当前查看器文本的当前选择的坐标范围。
- getSelection()——返回当前提供者的当前选择。
- getTextWidget()——返回查看器的文本部件。
- isEditable()——返回显示文本是否可被操作。
- refresh()——使用由查看器模型获取的信息完全刷新该查看器。
- setDocument(IDocument)——将给定文本设置为文本查看器模型，并正确地更新显示。
- setEditable(boolean)——设置可编辑模式。
- setInput(Object)——设置或清除当前查看器的输入。如果输入对象是一个IDocument的实例，该方法 TextViewer实现使用输入对象调用setDocument(IDocument)。如果输入对象不是IDocument的示例，则用null调用setDocument(IDocument)。
- setRedraw(boolean)——使当前文本查看器的重新绘制有效或无效。
- setSelectedRange(int, int)——将选择设置在指定的范围内。

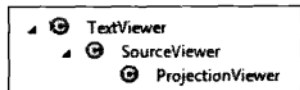


图5-11 TextViewer层次结构

- `setSelection(ISelection, boolean)`——为当前查看器设置一个新的选择，并可选择地使其可见。
- `setTextColor(Color)`——将给定颜色应用于当前查看器的选择。
- `setTextColor(Color, int, boolean)`——将给定颜色应用于当前查看器的指定部分。
- `setTextHover(ITextHover, String)`——设置当前查看器的文本悬浮为给定的内容类型。

以下示例创建一个包含样式文本的文本查看器（图5-12）。

```
import org.eclipse.jface.text.*;
import org.eclipse.swt.*;
import org.eclipse.swt.custom.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TextViewerExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Text Viewer Example");
        shell.setBounds(100, 100, 225, 125);
        shell.setLayout(new FillLayout());

        final TextViewer textViewer =
            new TextViewer(shell, SWT.MULTI | SWT.V_SCROLL);

        String string = "This is plain text \n"
            + "This is bold text \n"
            + "This is red text";
        Document document = new Document(string);
        textViewer.setDocument(document);

        TextPresentation style = new TextPresentation();
        style.addStyleRange(
            new StyleRange(19, 17, null, null, SWT.BOLD));
        Color red = new Color(null, 255, 0, 0);
        style.addStyleRange(
            new StyleRange(37, 16, red, null));
        textViewer.changeTextPresentation(style, true);

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

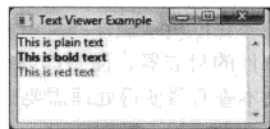


图5-12 TextViewer示例

在创建文本查看器后，创建了一个Document对象。它包含一个文本字符串，并被分配给查看器。然后，一个TextPresentation对象被创建以包含样式变化。添加了两个样式变化：一个设置一个范围的文本为粗体，另一个设置一个范围的文本为红色字体。StyleRange构造函数的第一个参数是样式将要应用于的字符串的第一个字符。第二个参数是应被样式影响的字符的个数。最后，样式对象被分配给查看器。

5.3 总结

在Eclipse插件开发中需要经常用到JFace查看器。列表查看器提供面向对象的用于基本的Eclipse部件的封装器，使处理高级别的域对象而不是处理简单字符串、数字和图像变得更容易。同样地，文本查看器使得处理需要更复杂文本样式的文本文档变得更容易。查看器将在第7章中进一步讨论。

参考文献

本书资源 (2.9节).

Gauthier, Laurent, "Building and Delivering a Table Editor with SWT/JFace,"

Mirasol Op'nWorks, July 3, 2003 (www.eclipse.org/articles/Article-Tableviewer/table_viewer.html).

Grindstaff, Chris, "How to Use the JFace Tree Viewer," Applied Reasoning, May 5, 2002 (www.eclipse.org/articles/treeviewer-cg/TreeViewArticle.htm).



第6章 命令与操作

命令（Command）与操作（Action）是用于完成同样任务的两种不同的API：声明与实现表现为菜单项和工具栏按钮的功能。操作API在Eclipse 3.0之前就已经出现了，而命令API直到Eclipse 3.3才刚刚成熟。在Eclipse 3.4中它进行了小幅度的改进。在一些地方，action API可能会被建议不使用，移至一个兼容性层，并在未来被移除。但是，很多Eclipse工具仍然在大量使用action API，还不用说基于Eclipse底层结构创建的第三方工具和IDE了。

正如Eclipse其他所有东西一样，命令与操作是通过不同的扩展点定义，因此新的功能可以在不同地方很容易地被添加至Eclipse框架。使用操作，对于每一个可出现在UI内部不同领域都有不同的扩展。然而，操作不能使表示和实现分离。相反，命令API将表示从实现分离出来。这是由以下方法实现的：通过提供一个扩展点来指定命令，另一个扩展点用于指定它在UI中出现的位置，第三个扩展点用于指定实现。这项功能和更丰富的声明语句语法，使得命令API比操作API更灵活。

本章的前半部分讨论如何使用命令API实现收藏夹插件，后半部分使用操作API来实现同样的功能（参见6.5节）。如果你使用了命令API实现了本章前半部分的所有内容，并用操作API实现本章后半部分的的所有内容，这样产生的收藏夹插件将具有重复的菜单和工具栏项。

6.1 命令

使用命令API声明并实现一个菜单或工具栏项包括：声明一个命令，至少一个对应该命令的菜单项和至少一个用于该命令的处理器（handler）（图6-1）。命令声明是将一个或多个菜单项与一个或多个处理器关联的抽象绑定点（图6-2）。

第一步是声明命令自身（参见下面的小节）。该命令表示一个UI功能概念，比如“粘贴”。在第一步中并不定义该功能将出现用户界面的哪个位置，或当用户选择该功能时将发生什么。菜单项声明（参见6.2节）定义了用户界面中该命令出现的位置，与该表示关联的文本和图像。处理器声明（参见6.3节）将命令与一个实现该命令行为的固定类关联起来。

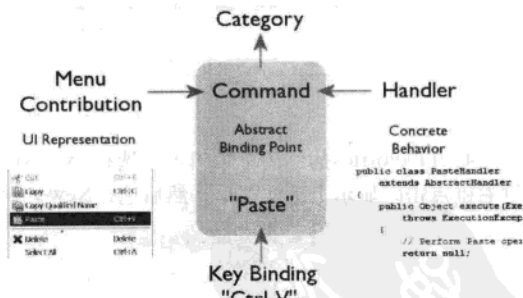


图6-1 命令概览

定义命令

添加菜单项或工具栏项的第一步是使用org.eclipse.ui.commands扩展点（图6-2）声明该功能的目标。通过使用该扩展点，你即声明了命令的类别（category），也声明了命令本身。类别对于轻松管理大量命令是十分有用的。

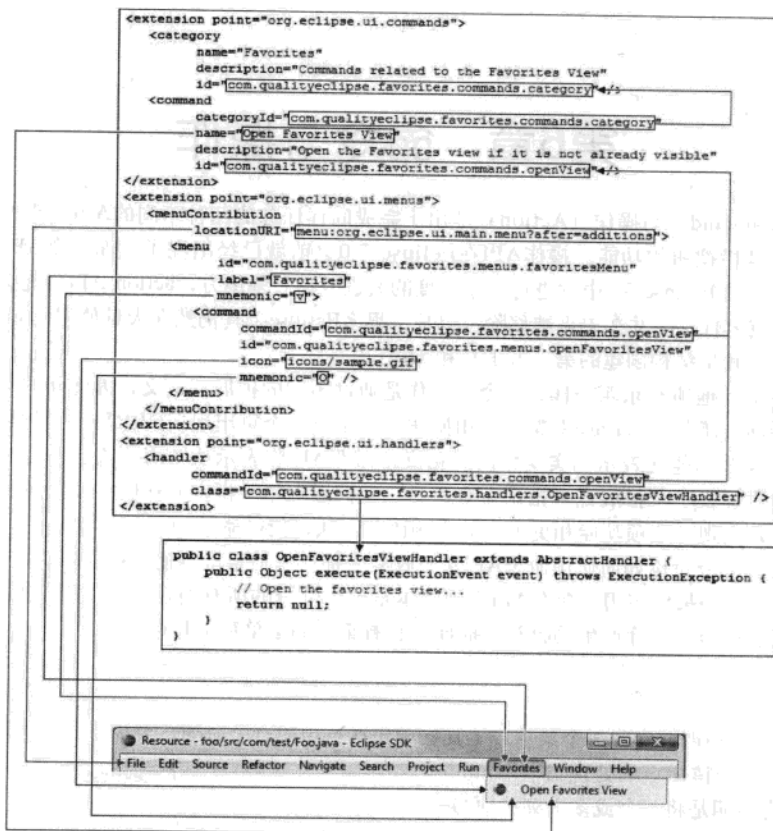


图6-2 命令扩展点概览

打开Favorites插件清单编辑器，选择Extensions选项卡，点击Add...按钮（图6-3）。你也可以通过右键点击以显示上下文菜单，然后选择New > Extension...命令以打开New Extension向导。

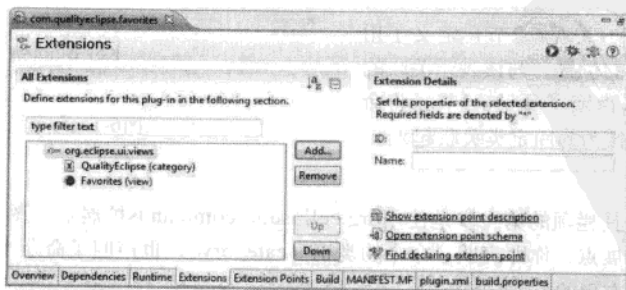


图6-3 清单文件编辑器的扩展项页面

从所有可用的扩展点（图6-4）列表中选择org.eclipse.ui.commands。如果你在列表中无法找到org.eclipse.ui.commands，取消选中Show only extension points from the required plug-ins单选框。点击Finish按钮以将该扩展点添加至插件的清单文件。

现在，让我们回到插件清单编辑器的Extensions页面，右键点击org.eclipse.ui.commands扩展项，并选择New > category。这将在插件清单中添加一个新的名为com.qualityeclipse.favorites.category1的命令类别。选择该新的类别将在编辑器的右侧显示属性。根据下列内容修改它们：

- id——“com.qualityeclipse.favorites.commands.category”

用于引用命令类别的唯一标识符。

- name——“Favorites”

类别的名称。

- description——“Commands related to the Favorites View”

类别的简要描述。

在添加了类别之后，再次右键点击扩展项org.eclipse.ui.commands并选择New > command。这将在插件清单中添加一个名为com.qualityeclipse.favorites.command1的新命令。选择该命令将在编辑器的右边显示属性。根据以下内容对属性做出更改：

- id——“com.qualityeclipse.favorites.commands.openView”

用于引用该命令的唯一标识符。

- name——“Open Favorites View”

命令的名称。该文本作为与没有明确声明标签属性的命令相关的任意menuContribution的标签属性（参见6.2.1节）。

- description——“Open the Favorites view if it is not already visible”

类别的简要描述。

- categoryId——“com.qualityeclipse.favorites.commands.category”

类别的唯一标识符，命令应只出现于该标识符中。

以同样的方式，声明另一个命令以用于向Favorites视图添加新的资源。

- id——“com.qualityeclipse.favorites.commands.add”

- name——“Add”

- description——“Add selected items to the Favorites view”

- categoryId——“com.qualityeclipse.favorites.commands.category”

当视图包含一个可以被添加至Favorites视图中的选项时，这些命令将会出现于视图的上下文菜单（参见6.2.5节）中。

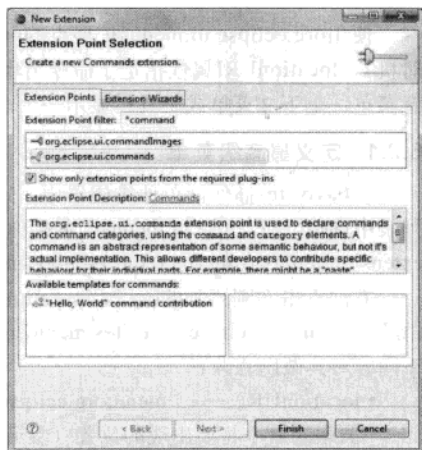


图6-4 显示扩展点的新建扩展项向导

提示 你可以添加命令参数以传递附加信息。参见15.5.5节以了解命令参数和15.5.4节以了解关于如何使用它们的示例。

6.2 菜单和工具栏添加项

使用org.eclipse.ui.menus扩展点的menuContribution元素定义命令出现在用户界面中的位置（和时机）。locationURI属性指定了命令出现的位置。使用menuContribution的这两个方面，你可是在任意菜单、上下文菜单或工具栏显示一个命令，并限制它的可见性为当它适用时。

6.2.1 定义最高级菜单

在Favorites插件清单编辑器中，选择Extensions选项卡，点击Add...按钮。从所有可用扩展点列表中选择org.eclipse.ui.menus。如果你无法在列表中找到org.eclipse.ui.menus，取消选中Show only extension points from the required plug-ins单选框。点击Finish按钮以将该扩展项添加至插件清单。

右键点击你刚添加的org.eclipse.ui.menus扩展项，并选择New > menuContribution。选择新的菜单项com.qualityeclipse.favorites.menuContribution1将在编辑器的右侧显示locationURI属性。根据以下内容对其做出修改：

- locationURI——“menu:org.eclipse.ui.main.menu?after=additions”标识用户界面中与该菜单项关联的命令将要出现的位置（参见6.2.9节以了解更多信息）。

右键点击你刚修改过的menuContribution，并选择New > menu。选择新的menu将在编辑器的右侧显示属性。根据下列内容对它们进行更改：

- label——“Favorites”
用于显示菜单的标签。
- id——“com.qualityeclipse.favorites.menus.favoritesMenu”
菜单的唯一标识符。

- mnemonic——“v”

标签中具有下划线的字符，表示键盘可访问性（参见6.4节）。

右键点击你刚修改的menu，并选择New > command。选择新的command以在编辑器的右侧显示属性。根据下列内容对它们做出更改：

- commandId——“com.qualityeclipse.favorites.commands.openView”

命令的标识符，当用户选择该菜单项时将触发该标识符。

- id——“com.qualityeclipse.favorites.menus.openFavoritesView”

该菜单项的唯一标识符。

- mnemonic——“O”

label中具有下划线的字符，表示键盘可访问性（参见6.4节）。

- icon——“icons/sample.gif”

包含图标图像文件的相对路径。该图标将显示于菜单项标签的左侧。

- label——留为空白

菜单项的文本。如果该项未被指定，作为替代，则将显示命令的名称。

6.2.2 添加至已有最高级菜单

上述步骤将一个全新的菜单添加至Eclipse菜单栏，如果你打算向Eclipse菜单栏中已有的菜单添加一个菜单项或子菜单，在locationURL中使用菜单的标识符而不是Eclipse菜单栏的标识符。在不同的org.eclipse插件中寻找menuContributions将出现locationURL，这样就可以将以下代码用于添加命

令至已有的Eclipse菜单:

```
menu:help?after=additions
menu:navigate?after=open.ext2
menu>window?after=newEditor
menu:file?after=open.ext
```

参见6.2.7节以了解关于添加菜单项至已有的最高级菜单的成熟示例。

6.2.3 定义最高级工具栏项

定义最高级工具栏项与定义最高级菜单项是十分类似的。如前面章节所述, 创建一个具有不同locationURI的menuContribution:

- locationURI——“toolbar:org.eclipse.ui.main.toolbar?after=additions”

标识用户界面中出现与该菜单项关联的命令的位置(参见6.2.9节以了解更多信息)。

右键点击新建的menuContribution并选择New > toolbar。选择新建的toolbar以在编辑器右侧显示属性。根据下列内容对它们做出更改:

- id——“com.qualityeclipse.favorites.toolbars.main”

工具栏的唯一标识符。

右键点击你刚修改的工具栏, 并选择New > command。选择新建的命令以在编辑器的右侧显示属性。根据下列内容对它们做出更改:

- commandId——“com.qualityeclipse.favorites.commands.openView”

命令的标识符, 当用户选择该工具栏项时触发该标识符。

- id——“com.qualityeclipse.favorites.toolbars.openFavoritesView”

该工具栏项的唯一标识符

- icon——“icons/sample.gif”

包含图标图像文件的相对路径。该图标将显示在工具栏中。

- tooltip——“Open the Favorites view”

当用户鼠标悬停于该工具栏项上显示的文本。

6.2.4 限制最高级菜单与工具栏项的可见性

最高级菜单、菜单项和工具栏项是一种很好的提供新功能的方法, 但当它们不可用时, 将很快把用户界面搞得杂乱无章(参见6.6.9节)。一个提供用户可配置可见性的方法是, 将菜单、菜单项和工具栏项分组为ActionSets。通过使用Customize Perspective对话框(图1-12), 用户可以控制特定ActionSet在当前透视图中是否可见。

要限制我们在前面章节创建的最高级菜单和工具栏项的可见性, 根据6.6.1节创建一个空ActionSet。然后根据下列内容, 将一个visibleWhen表达式(参见6.2.10节)添加至最高级菜单。重复相同步骤以同样显示最高级工具栏项的可见性。

右键点击菜单项(或工具栏项), 并选择New > visibleWhen。右键点击新建的visibleWhen元素, 并选择New > with。选择新建的with以在编辑器的右侧显示属性。根据下列内容对它们做出更改:

- variable——“activeContexts”

变量的名称将在运行时解析, 当评估子元素时使用。参见6.2.10节以了解已知变量。

右键点击with元素, 选择New > iterate。选择新建的iterate元素, 根据下列内容修改它的属性, 以使所有匹配的元素将触发表式以显示true, 而一个空集将显示false:

- operator——“or”
- ifEmpty——“false”

最后，右键点击iterate元素并选择New > equals，然后根据下列内容修改新建的equals元素的属性：

- value = “com.qualityeclipse.favorites.workbenchActionSet”

上面提到的空的ActionSet的标识符。

该新建的visibleWhen表达式仅当“activeContexts”集包含空的ActionSet的标识符时才为真。

6.2.5 定义基于选择的上下文菜单项

使用前面章节描述的同样机制，我们创建一个仅对于用户合适时才可见的上下文菜单和菜单项。我们从创建一个具有不同的locationURI的menuContribution开始：

- locationURI——“popup:org.eclipse.ui.popup.any?after=additions”

org.eclipse.ui.popup.any标识符通知Eclipse相关菜单和菜单项应在所有上下文菜单中出现（参见6.2.9节以了解更多细节）。

右键点击你刚做出修改的menuContribution，并选择New > menu。选择新建的菜单将在编辑器右侧显示属性。根据以下内容对它们做出修改：

- label——“Favorites”

右键点击你刚做出修改的菜单，选择New > command。选择新建的命令将在编辑器右侧显示属性。根据以下内容对它们做出修改：

- commandId——“com.qualityeclipse.favorites.commands.add”
- icon——“icons/sample.gif”

1. 限制上下文菜单项的可见性

如果我们在这里停止，则Favorites > Add菜单项将出现每一个上下文菜单中，即使该菜单是不适合的。我们需要菜单项根据资源和Java元素出现以使用户可以添加那些选中的对象至Favorites视图，但当用户右键点击Problems视图中的项时该菜单项是不可见的，这是由于该菜单项对于添加那些对象至Favorites视图是不合适的。为了完成该任务，我们将添加一个visibleWhen表达式（参见6.2.10节）。该表达式仅当一个或多个当前被选中的对象是IResource或IJavaElement的实例时才为真。

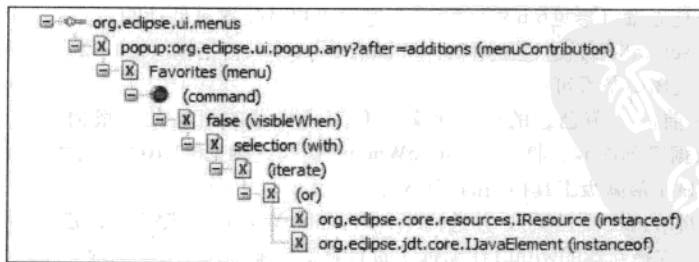


图6-5 具有visibleWhen表达式的弹出菜单

在上面的visibleWhen表达式（图6-5）中selection (with)元素解析当前被选中对象的集。iterate对象是selection (with)的后代，并根据集中的每一个对象来解析它的子元素。or元素和它的两个子元

素表示该对象必须是IResource或IJavaElement的实例。

要创建该visibleWhen表达式，从右键点击我们刚添加的菜单项开始。然后，选择New > visibleWhen。右键点击刚创建的visibleWhen元素，并选择New > with。选择新建的with元素以在编辑器的右侧显示属性。根据以下内容对它们做出修改：

- variable——“selection”

该变量的名称在运行时解析，当解析子元素时使用。参见6.2.10节以了解已知变量。

右键点击with元素，并选择New > iterate。选择新建的iterate元素，并根据以下内容修改它的属性。这样，任意匹配的元素将使表达式值为true。但是当空集合将使表达式值为false：

- operator——“or”
- iflmply——“false”

然后我们继续以同样方式创建表达式，以使iterate元素具有一个or子元素，该子元素又具有两个如下子元素：

- instanceof——value = “org.eclipse.core.resources.IResource”
- instanceof——value = “org.eclipse.jdt.core.IJavaElement”

当被测试的对象是IResource或IJavaElement的实例时，我们刚定义的or表达式和它的两个后代的值将为true。

在完成这项任务后，新的警告将出现于Problems视图中，标签为“Referenced class 'org.eclipse.jdt.core.IJavaElement' in attribute 'value' is not on the plug-in classpath”。要添加必要的插件至classpath并清除该警告，切换至Dependencies选项卡（图2-10），点击Add以添加一个新的必需插件，选择org.eclipse.jdt.core。

2. 创建新的propertyTester

我们的下一个目标是进一步降低在前一节添加的上下文菜单项的可见性。这一个目标可以通过测试一个对象是否已经被收藏夹集包含来实现。Eclipse提供了许多propertyTester用于解析选中对象的属性（参见6.2.10节），但没有提供我们在这种情况下需要的内容。我们必须创建一个新的propertyTester以完成我们的目标。

在Favorites插件清单编辑器中，选择Extensions选项卡，点击Add...按钮。从所有可用扩展点列表中选择org.eclipse.core.expressions.propertyTesters。如果你在该列表中无法找到org.eclipse.core.expressions.propertyTesters，取消选中Show only extension points from the required plug-ins单选框。点击Finish按钮以将该扩展项添加至插件清单。选择新建的propertyTester将在编辑器的右侧显示属性。根据下列内容对它们做出修改：

- id——“com.qualityeclipse.favorites.propertyTester”

该属性测试程序的唯一标识符。

- type——“java.lang.Object”

将要被该属性测试程序测试的对象的类型。只有这种类型的对象才会被传递至属性测试程序的test方法。

- namespace——“com.qualityeclipse.favorites”

定义属性将要被添加至命令空间的唯一id。

- properties——“isFavorite, notFavorite”

由该属性测试程序提供的由逗号分隔开的属性列表。

- class——“com.qualityeclipse.favorites.propertyTester.FavoritesTester”

提供测试行为的类的完全合格的名称。该类必须是public，并使用一个无参数的公共构造函数继承org.eclipse.core.expressions.PropertyTester。

一旦你已经输入上述内容指定的class属性，在属性字段左侧点击class标签。当打开New Java Class向导后，点击Finish以新建propertyTester类。FavoritesTester#test(...)方法应与以下内容类似：

```
public boolean test(Object receiver, String property, Object[] args,
    Object expectedValue) {
    if ("isFavorite".equals(property)) {
        // determine if the favorites collection contains the receiver
        return false;
    }
    if ("notFavorite".equals(property)) {
        // determine if the favorites collection contains the receiver
        return true;
    }
    return false;
}
```

当定义了收藏夹模型后，我们可以实现属性测试程序的test(...)方法（参见7.2.9节），但到此为止，该方法仍然只是一个实现框架。该框架表示被测试的对象不是收藏夹集的一部分。

提示 说明如何创建一个属性测试程序的屏幕截图可以在以下地址找到：<http://konigsberg.blogspot.com/2008/06/screencast-using-propertytesters-in.html>

3. 使用测试对象属性限制可见性

当我们完成新建propertyTester后，我们可以通过测试一个对象是否已经包含在收藏夹集中来进一步降低Add上下文菜单项的可见性。该项测试将被添加至本章之前内容添加的已有的instanceof元素（图6-6）中。

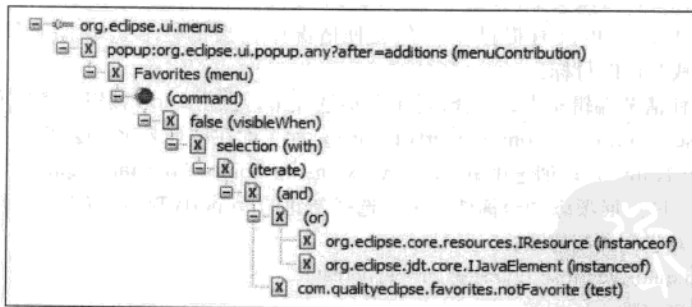


图6-6 具有visibleWhen表达式的弹出菜单

为了修改visibleWhen表达式，右键点击iterate元素并选择New > and。将or元素拖至and元素的第一个子元素上。右键点击刚创建的and元素并选择New > test，然后根据下列内容对测试元素的属性做出更改：

- property = “com.qualityeclipse.favorites.notFavorite”

将被测试的属性的完全合格的名称。该完全合格名称是该propertyTester命名空间后面紧跟由点

隔开，将要被测试的属性的名称。

- args = 留为空白

附加参数的由逗号隔开的列表。对于我们的属性测试程序，不需要附加参数。

- value = 留为空白

属性的预期值。我们的测试程序测试一个布尔值的属性，不需要预期值。

- forcePluginActivation = 留为空白

当插件还不是活动的时，强制激活该插件。参见下面的讨论以了解更多信息。

4. forcePluginActivation属性

无论何时用户右键单击一个选项，在前一节中定义的visibleWhen表达式将会被评估，已决定我们的Add菜单项是否应向用户呈现。如同6.2.10节描述的那样，visibleWhen表达式中的每一个元素将具有这三个值之一：true、false或not-loaded。如果总的visibleWhen表达式的值是true或not-loaded，那么相关联的菜单项和工具栏项将会是可见的。

我们的visibleWhen表达式包含了一个对我们的propertyTester的引用。该propertyTester在我们的插件中实现，因此我们的插件必须是活动的，以使表达式被正确的赋值为true或false。如果我们的插件不是活动的，那么引用我们的propertyTester的test元素将会被赋值为not-loaded。根据6.2.10节列出的逻辑，如果对象是IResource或IJavaElement的实例，那么visibleWhen表达式将会被赋值为not-loaded。否则，visibleWhen表达式将会被赋值为false。

所有这些工作的结果是，如果我们的插件没有被载入，我们的Add菜单项将会“乐观地”可见，而不需要强迫我们的插件被载入。这对于减少启动时间和降低内存占用是更好的选择，但是也存在更多地考虑准确性而不是其他因素的时候。在这种情况下，将test的forcePluginActivation属性设为true将使定义propertyTester的插件在没有被载入的情况下，立即被载入并激活。

6.2.6 定义视图相关菜单或工具栏项

为特定的视图添加命令与将命令添加至最高级Eclipse菜单栏或工具栏类似。简单地为该视图的上下文菜单、工具栏或下拉菜单确定标识符，并使用该标识符作为在locationURL中的id（参见6.2.9节）。一般地，视图标识符、上下文标识符，工具栏标识符和下拉菜单标识符都是一致的。

并不是说这样就是合适的，但是对于说明我们将Open Favorites View命令添加至Problems视图的下拉菜单、工具栏和上下文菜单是合适的。按照6.2.1节中描述的步骤创建三个新的menuContribution扩展性，并使用下列locationURL（为了了解更多有关locationURL的信息，参见6.2.9节）：

- pulldown —— “menu:org.eclipse.ui.views.ProblemView?after=additions”
- toolbar —— “toolbar:org.eclipse.ui.views.ProblemView?after=additions”
- context —— “popup:org.eclipse.ui.views.ProblemView?after=additions”

右键单击每一个新建的menuContributions并选择New > command。选择新建的command将在编辑器的右侧显示属性。根据6.2.1节列出的内容修改这些command属性。参见7.3.7节以了解另一个关于添加命令至上下文菜单的示例。

6.2.7 定义编辑器相关的菜单或工具栏项目

将命令添加至一个已有的编辑器的菜单或工具栏与添加最高级菜单项（参见6.2.1节）或工具栏项（参见6.2.3节）类似。区别在于前者必须具有一个visibleWhen表达式以使仅当特定类型的编辑器

是活动时才显示该命令。

作为一种展示该项技术的方法，当默认文本编辑器是活动的，我们将Open Favorites View命令添加至Window菜单。我们首先根据6.2.1节中描述的内容添加一个menuContribution，以下内容与6.2.1节中不同：

- locationURI——“menu:window?after=additions”

locationURL中的窗口标识符定义了新的菜单项应出现于Eclipse的Window菜单（参见6.2.9节以了解更多信息）。

右键点击新建的menuContribution并选择New > command，然后根据下列内容修改该新建命令的属性：

- commandId——“com.qualityeclipse.favorites.commands.openView”
- icon——“icons/sample.gif”

然后，右键点击新建的命令并选择New > visibleWhen。右键点击新建的visibleWhen元素并选择New > with，然后根据以下内容修改新建的with元素的属性：

- variable——“activeEditorId”

运行时解析并当给予元素赋值时使用的变量名称。在运行时，activeEditorId赋值给活动编辑器的标识符。参见6.2.10节以了解已知变量。

右键点击with元素并选择New > equals，然后根据以下内容修改新建的equals元素的属性：

value = “org.eclipse.ui.DefaultTextEditor”

一旦完成，该visibleWhen表达式仅当活动编辑器是Eclipse默认文本编辑器时为true。

6.2.8 动态菜单添加项

有些时候，你需要对添加的菜单项施加比之前提及的声明所能获取的更多控制。比如，如果你不知道将要被添加至菜单的菜单项或你需要一个具有一个检查标记紧挨着的菜单项，那么可以声明一个dynamic菜单添加项（参见14.3.7节以了解一个具有检查标记的菜单项）。

6.2.9 locationURI

正如之前几节所看到的那样，menuContribution的locationURI属性用于指定后续元素在用户界面中出现的位置（图6-7）。该属性被分解成三个清晰的部分：模式（scheme）、标识符（identifier）和参数列表（argument list）。

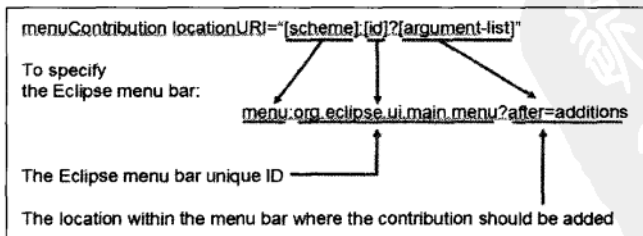


图6-7 LocationURI说明

scheme标识添加项将要被添加至的UI对象的类型。它可以是以下值的其中之一：

- menu——程序主菜单或视图下拉菜单。

- popup——视图或编辑器的上下文菜单。
- toolbar——程序主工具栏或视图中的工具栏。

locationURI的标识符或“id”定义了添加项将要被添加至的菜单、弹出项或工具栏的唯一标识符。比如，要添加项至视图的工具栏，指定模式为“toolbar”，视图的工具栏的标识符作为locationURI的id部分。规则在于，对于任意视图，它的标识符应与它的工具栏的标识符、上下文菜单的标识符和下拉菜单标识符一致。一些常用的标识符包括：

- org.eclipse.ui.main.menu——Eclipse主菜单的标识符。
- org.eclipse.ui.main.toolbar——Eclipse主工具栏的标识符。
- org.eclipse.ui.popup.any——任意上下文菜单的标识符。当该标识符以“popup”模式使用时，菜单和菜单项将在所有视图和编辑器的上下文菜单中可见。这与6.7节中讨论的上下文菜单中的操作十分类似。

locationURI的第三部分（也是最后一部分）是argument list。它允许给定菜单、弹出项或工具栏中的指定位置的细粒度定义。参数列表由可以是“before”或“after”的布局，一个等号（“=”），以及菜单、弹出项或工具栏的一些项的标识符组成。标识符也可以是“additions”，表示元素应当被放置于给定菜单、弹出项或工具栏的默认位置。

6.2.10 visibleWhen表达式

如同在前几节看到的那样，visibleWhen表达式控制特定菜单或工具栏项在用户界面中呈现的时机。visibleWhen表达式中的元素可以是以下三种状态之一：

- true
- false
- not-loaded

当创建表达式时，有几个你可以在组成表达式时可用的逻辑元素：

- and——值为true，仅当所有子元素的值为true时。
false，如果有至少一个子元素的值为false时。
not-loaded，在其他所有情况中。
- or——值为false，仅当所有子元素的值为false时。
true，如果有至少一个子元素的值为true时。
not-loaded，在其他所有情况中。
- not——值为true，仅当它的子元素的值为false时。
false，仅当它的子元素的值为true时。
not-loaded，仅当它的子元素的值为not-loaded。

一般地，当根据6.2.5节声明上下文菜单添加项时，作为父元素的with元素，将iterate元素作为子元素来测试当前选择的每一个对象。

- with——variable = “selection”

值为当前活动窗口中的当前活动视图或编辑器中的当前选择（一个集合）。参见6.2.10节以了解

Eclipse框架定义的其他变量的列表。

- count——value = integer, *, ?, +, !

当由父元素指定的集合包含指定数量的元素时，值为true。允许的通配符包括：

* = 任意数量的元素

? = 0个或1个元素

+ = 一个或多个元素

! = 没有元素

(参见7.3.7节以了解示例。)

- iterate——operator = “and/or”, ifEmpty = “true/false”

根据由父辈指定的集合中的每一个元素对子表达式进行赋值。操作符属性指定了为了将迭代元素赋值为true，是所有元素都必须赋值为true（操作符= “and”），还是只需要至少有一个元素值为true（操作符= “or”）。ifEmpty属性指定当集合为空时，由迭代元素返回的值。

当在一组对象中进行迭代时，一般地，你将探查每一个对象的一个或多个方面以决定是否应将visibleWhen表达式赋值为true，以使得相关联的用户界面元素可见。可以被赋值的方面包括：

- adapt——当由父元素指定的对象与指定类型匹配（参见21.3节）时，值为true。adapt元素可以具有子元素通过探测匹配的对象以进一步提炼表达式。适配表达式的后代通过使用and操作符联系起来。

提示 adapt元素目前是通过使用一个AdapterFactory辨认匹配的对象，但是不辨认实现IAdaptable接口的对象。为了了解更多细节，参见Bugzilla条目# 201743。

- equals——当由父元素指定的对象与指定值相等的话，值为true。字符串值将被转换为一个由6.2.10节中指定的Java元素。
- instanceof——当由父元素指定的对象是指定类型的实例时，值为true。
- systemTest——通过调用System.getProperty方法测试系统属性，并将结果与由值属性指定的值相比较。
- test——当对象的一个属性等于指定值时，值为true，否则为false。如果定义被引用的propertyTester的插件未被激活时，值为not-loaded。表达式中使用的对象来源于父元素，而一个propertyTester（参见6.2.10节）被用于执行实际的测试。与该元素相关的属性包括：
property = 将要测试的对象属性的名称。
args = 被传递至属性测试程序的附加参数。多个参数由逗号分隔。每一个独立元素被转换为一个Java元素，如同6.2.10节描述的那样。
value = 属性的预期值，当属性是一个布尔属性时，可以被省略。值属性根据6.2.10节中列出的那样转换为一个Java元素。
forcePluginActivation = 一个表示当需要时，添加项属性测试程序的插件是否应被载入的标记。不要指定该属性，除非特别需要（参见6.2.5节）。

1. 使用变量

你可以指定with元素中的其他变量，如同6.2.7节那样。with元素可能的变量如下所示。为了了解更多变量，参见org.eclipse.ui.ISources类。如果你确实感觉很冒险，请搜索org.eclipse.core.expressions.EvaluationContext#addVariable(...)的调用者。

- “activeContexts”——值为一个ActionSet和上下文标识符的集合。该标识符在6.2.4节中讨论。
- “activeEditorId”——值为当前活动编辑器的标识符。该标识符在6.2.7节中讨论。
- “activePartId”——值为当前活动部分的标识符（参见7.3.2节）。
- “selection”——值为当前活动窗口中的当前活动部分（视图或编辑器）的当前选择（一个集合），如同上面和6.2.5节中讨论的那样。

提示 为了了解更多关于命令表达式和在with元素中使用的变量，参见http://wiki.eclipse.org/Command_Core_Expressions。

2. 字符转换为Java元素的约定

当给一个包含插件清单中指定的字符串的表达式赋值（参见6.2.10节）时，以下规则被使用于在比较前将该字符串转换为Java元素：

- 字符串“true”被转换为Boolean.TRUE。
- 字符串“false”被转换为Boolean.FALSE。
- 如果字符串包含一个圆点，则解释程序试图将值转换为一个Float对象。如果失败了，字符串将作为一个java.lang.String对待。
- 如果字符串仅由数字组成，那么解释程序将值转换为一个Integer对象。
- 在其他所有情况，字符串将作为java.lang.String对待。
- 字符串转换为Boolean，Float或Integer的约定可以通过使用单引号包围字符串来隐藏。比如，属性值“true”将被转换为字符串“true”。

3. propertyTester

propertyTester是用于给特定对象的状态赋值的表示式框架的扩展。当用于与with或iterate表达式联合（参见6.2.10节）时，propertyTester接受表达式指定的对象和将被测试的属性名作为参数传递给它的test方法（参见6.2.5节），并返回一个布尔值以表示对象是否处于预期状态。

Eclipse提供了许多属性测试程序（参见下面的列表）以和test表达式一起使用（参见6.2.5节）你也可以定义你自己的属性测试程序（参见6.2.5节）。搜索所有对于org.eclipse.core.expressions.propertyTesters扩展点的扩展以了解一个更完整的属性测试程序列表。所有的属性测试程序必须继承org.eclipse.core.expressions.PropertyTester。因此，为了了解更多细节，打开一个类型层次结构视图并浏览它的不同子类。

org.eclipse.core.runtime.Platform

- org.eclipse.core.runtime.isBundleInstalled——当一个具有等于第一个参数的标识符的集合被载入时，值为true。
- org.eclipse.core.runtime.product——当当前产品的标识符等于指定字符串时，值为true。
- org.eclipse.core.resources.IResource
- org.eclipse.core.resources.name——当IResource的名称与指定样式（预期值）匹配时，值为true。
- org.eclipse.core.resources.path——当IResource的路径与指定样式（预期值）匹配时，值为true。
- org.eclipse.core.resources.extension——当IResource的扩展项与指定样式（预期值）匹配时，值为true。

- `org.eclipse.core.resources.readOnly`——当`IResource`的只读标记等于指定布尔值时，值为`true`。
`org.eclipse.core.resources.IFile`
- `org.eclipse.core.resources.contentType`——当`IFile`的内容类型标识符等于预期值时，值为`true`。
`org.eclipse.core.resources.IProject`
- `org.eclipse.core.resources.open`——当`IProject`打开时，值为`true`。
`org.eclipse.ui.IWorkbench`
- `org.eclipse.ui.isActivityEnabled`——当具有等于第一个参数的标识符的活动可以被找到，且当前是可用的时，值为`true`。
- `org.eclipse.ui.isCategoryEnabled`——当具有等于第一个参数的标识符的范围可以被找到，且当前是可用的时，值为`true`。
`org.eclipse.ui.IWorkbenchWindow`
- `org.eclipse.ui.workbenchWindow.isPerspectiveOpen`——当活动工作台窗口页是一个透视图时，值为`true`。

6.3 处理器

一个命令要发挥作用，它必须具有相关的一些行为。使用`org.eclipse.ui.handlers`扩展点，你可以将一个或多个实现命令行为的固定类和命令本身关联起来。作为替代，你也可以使用`IHandlerService`从程序上将处理器和一个命令关联起来，就如同7.3.6节中描述的那样。

绝大部分处理器具有`activeWhen`和`enabledWhen`表达式用于指定适合执行处理器的时机（参见7.3.7节以了解相关示例）。这些表达式与6.2.10节中描述的`visibleWhen`表达式具有类似的格式。如果一个处理器不具有任何表达式，它将被作为默认处理器（default handler）对待。默认程序仅当没有其他处理器的所有条件都被满足时可用。

如果用户选择一个命令，并且两个或两个以上的处理器具有被满足的条件，那么条件将会被比较。解决办法是选择一个条件更明确或更本地化的处理器。要完成该任务，由条件引用的变量将被监视，引用更明确变量的条件“胜出”（参见`org.eclipse.ui.ISources`）。如果这样仍不能解决冲突，那么将没有处理器是活动的。当有两个或两个以上默认处理器时，也会引发冲突。

提示 我们强烈建议指定一个`activeWhen`表达式以防止不必要的插件载入。对于将要被载入的处理器（和定义它的插件），命令必须由用户选定，而且`activeWhen`和`enabledWhen`表达式必须被满足。

在Favorites插件清单编辑器中，选择Extensions选项卡，并点击Add...按钮。从所有可用扩展点列表中选择`org.eclipse.ui.handlers`。如果在列表中无法找到`org.eclipse.ui.handlers`，取消选中Show only extension points from the required plug-ins单选框。点击Finish按钮以将该扩展项添加至插件清单。

在插件清单编辑器的Extensions页中右键点击`org.eclipse.ui.handlers`扩展项并选择New > handler。选择新建的`com.qualityeclipse.favorites.handler1`将在编辑器的右侧显示属性。根据下列内容对它们做出修改：

- `commandId`——“`com.qualityeclipse.favorites.commands.openView`”
用于引用命令的唯一标识符。

- class——“com.qualityeclipse.favorites.handlers.OpenFavoritesViewHandler”

用于执行操作的org.eclipse.core.commands.IHandler对象（参见6.3.1节）。该类将使用它的无参数构造函数初始化，但可以使用IExecutableExtension接口指定参数（参见21.5节中指定的类型）。

提示 如果你拥有一个已有的操作（参见7.3.4节），使用org.eclipse.jface.commands.ActionHandler将它转换为一个IHandler的实例。

当该类不存在时，点击该属性的文本框左侧的class:标签将打开新建Java类向导。当该类已经存在时，将打开Java编辑器。

创建新的IHandler

处理器包含了与命令相关的行为。以下是几种可以将处理器与命令相关联的方法：

- 在class字段中输入处理器的完全合格类名称。
- 点击class字段左侧的class:标签以创建一个新的处理器类。
- 点击class字段右侧的Browse...按钮以选择一个已有的处理器。

当你还没有完成创建一个该命令的处理器类，Eclipse将生成一个可以被自定义的处理器类。选择在上一节中创建的处理器并点击class字段左侧的class:标签为操作类打开New Java Class Wizard。

在创建了类，打开编辑器之后，根据下列内容修改execute(...)方法，当用户选择操作时，收藏夹视图将会打开。org.eclipse.ui.handlers.HandlerUtil类提供了几种来源于execute(...)方法内部的有用的方法。

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {

    // Get the active window
    IWorkbenchWindow window = HandlerUtil
        .getActiveWorkbenchWindowChecked(event);
    if (window == null)
        return null;
    // Get the active page
    IWorkbenchPage page = window.getActivePage();
    if (page == null)
        return null;

    // Open and activate the Favorites view

    try {
        page.showView(FavoritesView.ID);
    } catch (PartInitException e) {
        FavoritesLog.logError("Failed to open the Favorites view", e);
    }
    return null;
}
```

在本书写作时，Eclipse API指定execute(...)方法的返回值为将来的使用而保留，并且必须为null。然后，向FavoritesView类中添加一个表示用于打开Favorites视图的唯一标识符的常量。

```
public static final String ID =
    "com.qualityeclipse.favorites.views.FavoritesView";
```

以同样的方式，我们需要为6.1.1节中介绍的com.qualityeclipse.favorites.commands.add命令添加

一个处理器。重复以上步骤以添加一个新的AddToFavoritesHandler处理器。

提示 你可以添加命令参数以传递附加信息。参见15.5.5节以了解在处理器中处理命令参数的示例。

6.4 键绑定

命令并不引用键绑定，然而键绑定是单独声明的，并引用命令。这样允许多个键绑定与同一个命令关联。比如，保存文件内容的默认快捷键是Ctrl+S，但切换至Emacs配置后，保存的快捷键变为Ctrl+X Ctrl+S。

为了为Add命令添加一个键绑定，创建一个新的org.eclipse.ui.bindings扩展项（参见6.1.1节以了解如何添加扩展项的示例），然后右键点击并选择New > key。为新建的键绑定输入以下属性。一旦完成后，收藏夹键绑定就将出现在Keys首选项页（图6-8），而当聚焦于文本编辑器时，Ctrl+Shift+A将触发Add命令。

- commandId——“com.qualityeclipse.favorites.commands.add”

由键绑定触发的命令。

- contextId——“org.eclipse.ui.textEditorScope”

对于用户可用的键绑定所在的上下文环境。一些预定义的范围包括：

- org.eclipse.ui.contexts.window——工作台窗口
- org.eclipse.ui.textEditorScope——文本编辑器
- org.eclipse.ui.contexts.dialog——对话框
- org.eclipse.jdt.ui.javaEditorScope——Java编辑器
- org.eclipse.debug.ui.debugging——调试视图
- org.eclipse.debug.ui.console——终端视图

新的上下文环境可以使用org.eclipse.ui.contexts扩展点进行定义。如果没有指定contextId，它将使用默认值org.eclipse.ui.contexts.window。

- schemeId——“org.eclipse.ui.defaultAcceleratorConfiguration”

包含键绑定的用户可选的模式。一般地，键绑定被添加至默认Eclipse配置，但作为替换的键绑定，如“org.eclipse.ui.emacsAcceleratorConfiguration”，可以被添加至其他配置文件。可以通过在org.eclipse.ui.bindings扩展项中声明一个新的scheme元素来定义新的模式。

- sequence——“Ctrl+Shift+A”

被分配给该命令的键位序列。键位序列由一个或多个键击组成。一个键击由键盘上的一个按键，可选地与一个或多个下列限定键一同按下：Ctrl、Alt、Shift、Command、M1（对应于特定平台的Ctrl或Command）、M2（Shift）和M3（对应于特定平台的Alt或Option）。键击由空格隔开，而限定键由“+”字符隔开。X后面跟着按下控制键和按下S是“Ctrl+X Ctrl+S”。特殊键由以下内容表示：ARROW_DOWN、ARROW_LEFT、ARROW_RIGHT、ARROW_UP、BREAK、BS、CAPS_LOCK、CR、DEL、END、ESC、F1、F2、F3、F4、F5、F6、F7、F8、F9、F10、F11、F12、F13、F14、F15、FF、HOME、INSERT、LF、NUL、NUM_LOCK、NUMPAD_0、NUMPAD_1、NUMPAD_2、NUMPAD_3、NUMPAD_4、NUMPAD_5、NUMPAD_6、NUMPAD_7、NUMPAD_8、NUMPAD_9、NUMPAD_ADD、NUMPAD_DECIMAL、NUMPAD_DIVIDE、NUMPAD_ENTER、NUMPAD_EQUAL、

NUMPAD_MULTIPLY、NUMPAD_SUBTRACT、PAGE_UP、PAGE_DOWN、PAUSE、PRINT_SCREEN、SCROLL_LOCK、SPACE、TAB和VT。对于一些通用的特殊键，有不同的名称。比如，ESC和ESCAPE是一致的。CR、ENTER和RETURN是一致的。

其他没有在Favorites示例中使用的键绑定属性包括：

- **locale**——表示键绑定只定义用于特定范围的可选属性。范围根据java.util.Locale中声明的格式指定。
- **platform**——表示键绑定只定义用于特定平台的可选属性。平台属性的可能值是由org.eclipse.swt.SWT.getPlatform()返回的可能值的集合。

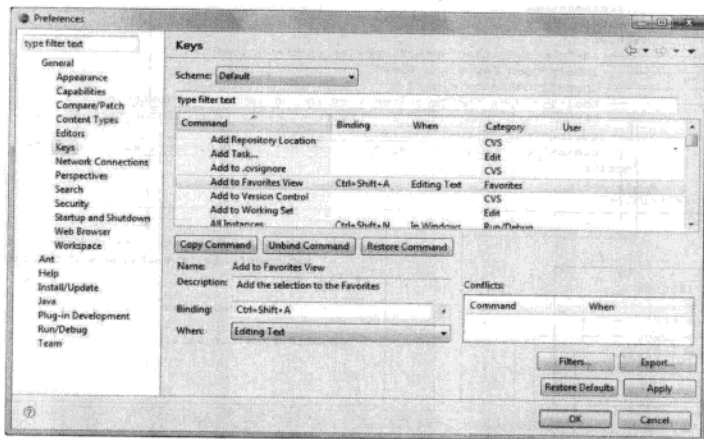


图6-8 显示Favorites键绑定的键位首选页

6.5 IAction与IActionDelegate

Eclipse操作由几部分组成，包括插件清单中的操作的XML声明、由Eclipse UI初始化并代表操作的IAction对象和由包含执行操作的代码的插件库定义的IActionDelegate（图6-9）。

IAction对象的这种分离，由Eclipse用户界面根据插件清单和插件库定义的IActionDelegate来定义和初始化，允许Eclipse在菜单或工具栏中表示操作，仅当用户选择一个特定菜单项或在工具栏上点击时才载入包含该操作的插件。这种实现又一次体现了Eclipse的总体主题之一：惰性插件初始化。

IActionDelegate有几个有趣的子类型。

- **IActionDelegate2**——为操作代表提供生命周期事件。如果你正在实现IActionDelegate且需要附加信息，如在操作代表被释放前何时进行清理，那么实现IActionDelegate2作为替代。此外，实现了IActionDelegate2的操作代表将调用runWithEvent（IAction，Event）而不是run（IAction）。
- **IEditorActionDelegate**——为与编辑器关联的操作代表提供生命周期事件（参见6.9.3节）。
- **IObjectActionDelegate**——为与上下文菜单关联的操作代表提供生命周期事件（参见6.7.3节）。
- **IViewActionDelegate**——为与视图关联的操作代表提供生命周期事件（参见6.8.3节）。

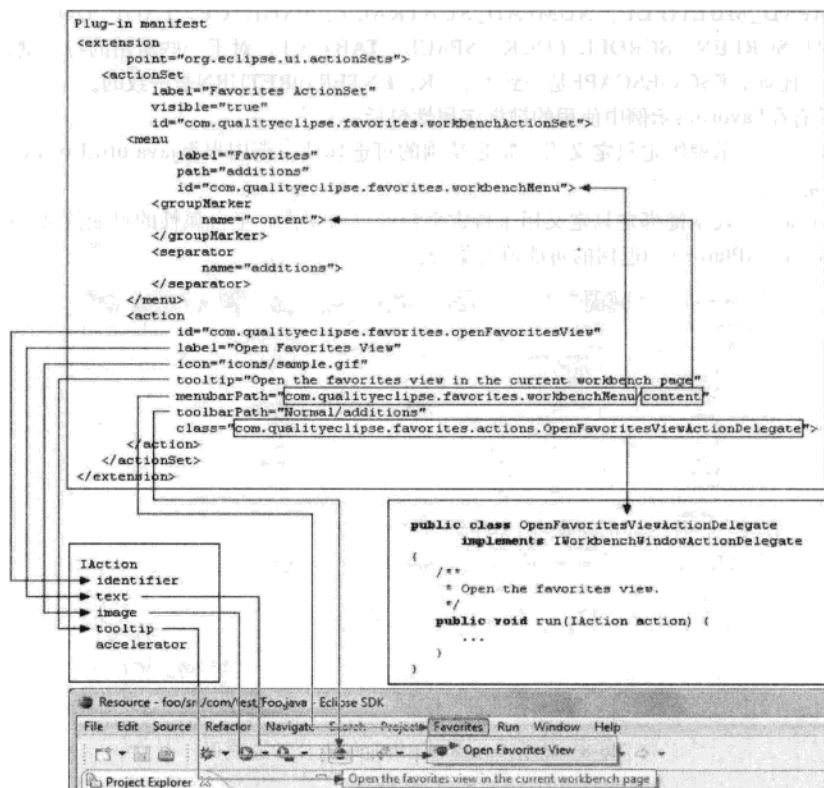


图6-9 Action与IActionDelegate

- `IWorkbenchWindowActionDelegate`——为与工作台窗口菜单栏或工具栏关联的操作代表提供生命周期事件。

6.6 工作台窗口操作

操作出现于何时何地取决于扩展点和用于定义操作的过滤器。本节讨论使用 `org.eclipse.ui.actionSets` 扩展点 (图6-9) 将一个新的菜单添加至工作台菜单栏和一个新的按钮添加至工作台工具栏。这些操作与具有值为 “`org.eclipse.ui.main.menu`” 的 `locationURI` `id` (参见6.2.9节) 的菜单添加项十分相似。

当用户选择菜单项和工具栏按钮时, 均能打开 Favorites 视图。用户可以已经打开 Favorites 视图 (如2.5节中描述的那样), 但最高级菜单通过提供一种简便的方法以找到收藏夹视图来突出显示该新产品。

提示 最高级菜单是向用户突出显示新产品的很好的方法, 但是请务必阅读6.6.9节以了解这种实现中易犯的错误。

6.6.1 定义工作台窗口菜单

为了创建出现于工作台菜单栏的新菜单，你需要在Favorites插件清单中创建一个actionSet扩展项以描述新的操作。该声明必须描述新菜单的位置和内容，并引用执行操作的操作代表类。

打开Favorites插件清单编辑器，选择Extensions选项卡，点击Add...按钮（图6-3）。你也可以通过右键点击，显示上下文菜单以打开New Extension向导，然后选择New > Extension...命令。

从所有可用扩展点列表中选择org.eclipse.ui.actionSets（图6-10）。如果你在列表中无法找到org.eclipse.ui.actionSets，取消选中Show only extension points from the required plug-ins单选框。点击Finish按钮以将该扩展项添加至插件清单。

现在，让我们回到插件清单编辑器的Extensions页。右键点击org.eclipse.ui.actionSets扩展项并选择New > actionSet。这将立即在插件清单中添加一个名为com.qualityeclipse.favorites.actionSet1的操作集。选择该新建操作集将在编辑器的右侧显示其属性。根据以下内容对他们做出更改：

- id——“com.qualityeclipse.favorites.workbenchActionSet”
用于引用操作集的唯一标识符。
- label——“Favorites ActionSet”
出现在Customize Perspective对话框的文本。
- visible——“true”

决定操作集是否在初始时是可见的。用户可以通过选择Window > Customize Perspective..., 在Customize Perspective对话框中展开Other节点来显示或隐藏操作集，并选中或不选中列出的不同操作集。

然后，通过右键点击你刚添加的操作集并选择New > menu来添加一个将出现在工作台菜单栏的菜单。请注意，当树的选择改变时，新的操作集的名称变为Favorites ActionSet。选择新的菜单并根据以下内容设置它的属性（参见图6-11）：

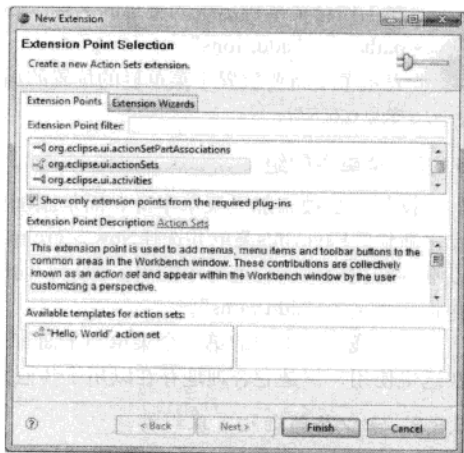


图6-10 显示扩展点的新扩展项向导

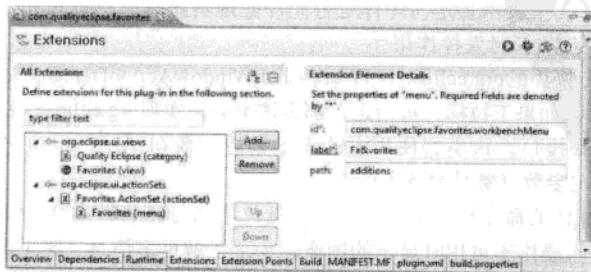


图6-11 显示收藏夹菜单的属性的扩展项页

- id——“com.qualityeclipse.favorites.workbenchMenu”
用于引用该菜单的唯一标识符。
- label——“Fa&vorites”
在工作台菜单栏出现的菜单的名称。“&”用于键盘快捷键（参见6.10.2节）。
- path——“additions”
表示菜单将要放置于菜单栏的位置的插入点。为了了解更多关于“additions”和插入点的信息，参见6.6.5节。

6.6.2 菜单中的组

操作不是被添加至菜单本身，而是添加至菜单中的组。因此，我们首先必须定义一些组。右键点击新建的Favorites菜单并选择New > groupMarker。选择新的groupMarker并将name改为“content”以唯一标识Favorites菜单中的组。然后，向Favorites菜单添加第二个组。这次选择New > separator并将其命名为“additions”。

分割线组在组中的第一个菜单项上面有一条水平线，而groupMarker没有任何线。additions组不在此处使用，但是它合理地存在以用于其他插件需要向插件菜单添加项操作。

6.6.3 定义菜单项和工具栏按钮

最后，是时候定义出现于Favorites菜单和工作台工具栏的操作的时候了。右键点击Favorites ActionSet并选择New > action。选择该新的操作并输入下列值：

- id——“com.qualityeclipse.favorites.openFavoritesView”
用于引用操作的唯一标识符。
- label——“Open Favo&rites View”
在收藏夹菜单中出现的文本。“&”用于键盘快捷键（参见6.10.2节）。
- menubarPath——“com.qualityeclipse.favorites.workbenchMenu/content”
表示操作出现于菜单的位置的插入点（参见6.6.5节）。
- toolbarPath——“Normal/additions”
表示按钮出现于工具栏的位置的插入点（参见6.6.5节）。
- tooltip——“Open the favorites view in the current workbench page”
当鼠标悬停于工作台工具栏中的操作图标时出现的文本。
将在后续几节讨论的其他属性，包括以下内容：
- allowLabelUpdate——表示重定向操作是否允许处理器覆盖它的标签和工具提示的可选属性。
它仅当重定向属性是true时发挥作用。
- class——用于执行操作的org.eclipse.ui.IWorkbenchWindowActionDelegate表示将在后续章节中讨论（参见6.6.6节）。如果下拉样式被指定，那么该类必须实现org.eclipse.ui.IWorkbenchWindow-PullDownDelegate接口。该类将使用它的无参构造函数初始化，但也可以使用IExecutable-Extension接口指定参数（参见21.5节）。
- definitionId——操作的命令标识符，允许一个键位序列与它关联（参见6.10.1节）。
- disabledIcon——当操作不可用时显示的图像。为了了解更多信息，参见6.6.4节。
- enablesFor——表示操作将被激活的时机的表达式（参见6.7.2节）。如果值为空，那么操作将一直可用，除非通过IAction接口在程序中将其覆盖。

- `helpContextId`——与操作关联的帮助上下文环境的标识符（在第15章中提及）。
- `hoverIcon`——当指针悬停于操作上而不点击时显示的图像。为了了解更多信息，参见6.6.4节。
- `icon`——相关的图像。为了了解更多信息，参见6.6.4节。
- `retarget`——重定向该操作的可选属性。当它的值为`true`时，视图和编辑器部分可能为该操作提供处理器。该处理器使用标准机制以用于在网站中使用该操作标识符设置一个全局操作处理器（参见8.5.2节）。如果该属性为`true`，则不应提供类属性。
- `state`——设置初始状态为`true`或`false`以决定操作使用`radio`还是`toggle`样式。
- `style`——定义操作的可视界面的属性。它具有以下值之一：
 - `push`——普通菜单或工具栏项（默认样式）。
 - `radio`——一个单选按钮样式的菜单或工具栏项。一组都具有按钮样式的项中同一时间只有一项可以被使用。参见`state`属性。
 - `toggle`——选中菜单项或切换工具项。参见`state`属性。
 - `pulldown`——子菜单或下拉工具栏菜单。参见`class`属性。

6.6.4 操作的图像

然后，将图标与将出现于工作台工具栏的操作相关联。选择在之前一节添加的Open Favorites View操作，点击出现于`icon`字段右侧的Browse...按钮。在出现的对话框中，展开树并从`icons`文件夹中选择`sample.gif`项（图6-12）。点击OK按钮，`icons/sample.gif`将会出现于`icon`字段。

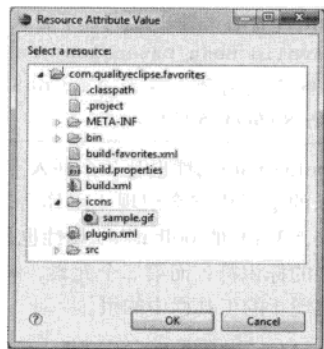


图6-12 用于选择图标的资源属性值对话框

`icon`字段和`plugin.xml`中的路径是相对于插件安装目录的。包括`hoverIcon`和`disabledIcon`在内的其他用于指定图像的图像属性当鼠标悬停于工具栏按钮和操作不可用时分别被使用。

创建自己的图标 有几个可以用于创建和修改图像的程序，包括Jasc的Paint Shop Pro和Adobe的Photoshop Elements。使用这些中的一个程序，你可以从头开始创建一个图标，或首先由Eclipse提供的众多图标中的一个（要开始这项工作，参见`org.eclipse.ui`或`org.eclipse.jdt.ui` JAR文件中的`\icons\full`目录）。图标一般是*.gif文件，并且是透明的。

6.6.5 插入点

由于Eclipse是由多个插件组成（每一个插件都提供不同的操作，但在创建时彼此并不相互知晓），而操作或子菜单出现于父辈的位置直到运行时才知晓。即使在执行过程中，当用户改变选择时，位置也会因为同级别对象的操作而被添加或移除。基于这种原因，Eclipse使用标识符（identifier）来引用菜单、组或操作，以及用于指定菜单或操作出现位置的路径（即插入点）。

每一个插入点由一个或两个由斜杠隔开的标识符组成。这些标识符表示操作将会被放置于父辈（在这里是一个菜单）和组的位置。比如，Open Favorites View操作的`menubar`属性（参见6.6.3节和图6-1）是由两个由斜杠隔开的元素组成。

第一个元素`com.qualityeclipse.favorites.workbenchMenu`标识了Favorites菜单，而第二个元素

content标识了Favorites菜单中的组。在某些情况下，如当父辈是工作台菜单栏或视图的上下文菜单时，该父辈被隐藏，因此插入点只指定组。

一般地，插件为其他插件提供帮助以通过定义一个标签为“additions”的空组来添加新的操作至它们自己的菜单。新的操作将会出现于该组中。“additions”标识符实际上在整个Eclipse中就是一个标准，它表示新操作或菜单将出现的位置。该标识符将被这些操作或菜单作为IWorkbenchActionConstants.MB_ADDITIONS常量所包含。比如，Favorites菜单指定了一个path属性（参见6.6.1节）。该属性的值为“additions”，使得Favorites菜单出现于Window菜单的左侧。因为Window菜单的标识符是window，而如果收藏夹菜单的path属性被设为“window/additions”，那么Favorites菜单将作为Window菜单自身的子菜单出现，而不是出现于工作台菜单栏中。

嵌套actionSet问题 在添加项于不同actionSet定义的菜单的actionSet中定义一个操作可能在Eclipse日志文件中生成以下错误：

```
Invalid Menu Extension (Path is invalid): some.action.id
```

要解决该问题，在两个actionSet中都定义该菜单。为了了解更多信息，参见Bugzilla条目#36389和#105949。

toolbarPath属性也是一个插入点，并具有和menubarPath属性一致的结构。但是，toolbarPath属性表示的是操作将会出现于工作台工具栏而不是菜单栏的位置。比如，Open Favorites View操作（参见6.6.3节）的toolbarPath属性也是由斜杠隔开的两个元素组成：第一个元素，Normal，是工作台菜单栏的标识符；而第二个元素，additions，是操作将要出现于的工具栏中的组。

6.6.6 创建操作代表

距离完成操作只剩下操作代表（action delegate）了。操作代表包含与操作关联的行为。以下内容是可以用于指定与操作关联的操作代表的几种方法。

- 在class字段输入操作代表的完全合格类名。
- 点击class字段左侧的class:标签以创建一个新的操作代表类。
- 点击class字段右侧Browse...按钮以选择一个已有的操作代表。

如果你还没有为操作完全创建好类，Eclipse将会为你生成一个可以让你自定义的类。选择Open Favorites View操作并点击class字段左侧的class:标签，这样将为操作类打开新建Java类向导（图6-13）。

在Package字段输入“com.qualityeclipse.favorites.actions”，在Name字段输入“OpenFavoritesViewActionDelegate”。点击Finish按钮以生成新的操作代表并为该新建的类打开一个编辑器。

当创建了类并打开了编辑器之后，根据以下内容修改该类的内容。这样，收藏夹视图将会当用户选择该操作时打开。我们从添加一个新字段和修改init()方法以缓存该操作代表运行所在窗口开始。

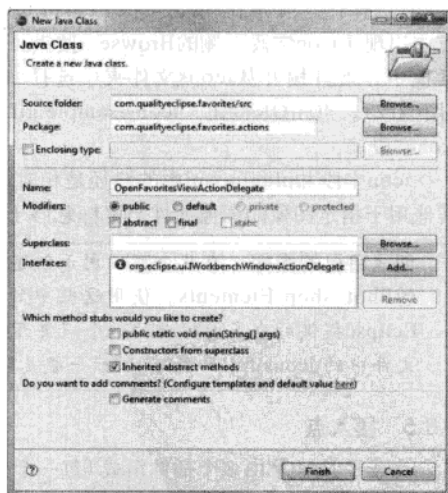


图6-13 操作类的新建Java类向导

```
private IWorkbenchWindow window;

public void init(IWorkbenchWindow window) {
    this.window = window;
}
```

然后，向FavoritesView类添加一个表示用于打开Favorites视图的唯一标识符的常量。

```
public static final String ID =
    "com.qualityclipse.favorites.views.FavoritesView";
```

最后，修改OpenFavoritesViewActionDelegate类的run()方法以真正打开Favorites View。

```
public void run(IAction action) {

    // Get the active page.
    if (window == null)
        return;

    IWorkbenchPage page = window.getActivePage();
    if (page == null)
        return;

    // Open and activate the Favorites view.
    try {
        page.showView(FavoritesView.ID);
    }
    catch (PartInitException e) {
        FavoritesLog.logError("Failed to open the Favorites view", e);
    }
}
```

1. selectionChanged方法

插件清单中的操作声明提供了操作的初始状态，而操作代表中的selectionChanged()方法提供了动态调整属性的方法。这些属性包括：状态、可用状态和使用IAction接口的操作的文本。

比如，enablesFor属性（参见6.7.2节）用于指定对象数目。这些对象用于选择一个操作并使其可用。而这项使能操作的进一步改进可以通过实现selectionChanged()方法来进行。该方法可以获取当前选择并在需要时调用IAction.setEnabled()方法以更新操作的使能状态。

为了可以调用操作代表的selectionChanged()方法，你需要在你视图的createPartControl()方法中调用getViewSite().setSelectionProvider(viewer)。

2. run方法

当用户选择操作和监视将要执行的任务时将会调用run()方法。与selectionChanged()方法类似，IAction接口可以用于改变依赖任务输出的操作的状态。

必需的保护代码 请留意以下情况：如果没有载入插件，并且用户选择菜单选项使得插件将要被载入，selectionChanged()有可能不会在run()方法之前被调用。这样，run()方法仍然需要合适的保护代码。另外，run()方法是在主UI线程中执行，因此，请考虑将长运行时间的任务置于后台线程中（参见21.8节）。

6.6.7 手动测试新建操作

测试你刚完成的更改包括根据第2章启动Runtime Workbench。如果Favorites菜单没有在Runtime

Workbench菜单栏中出现,或无法在工具栏中找到Favorites图标,可以尝试以下方法:

- 通过选择Window > Customize Perspective...打开Customize Perspective对话框使操作集可用。在该对话框中,选择Commands选项卡,找到Favorites ActionSet,并确认它是选中的(图6-14)。
- 使用Window > Reset Perspective重新初始化该透视图。
- 关闭并重新打开该透视图。
- 如果没有其他方法生效,那么尝试在启动Runtime Workbench之前清理工作区数据。要执行该任务,在启动菜单中选择Run...,选择Favorites启动配置文件,并选中Clear workspace data before launching单选框。点击Run按钮以启动运行时工作台。

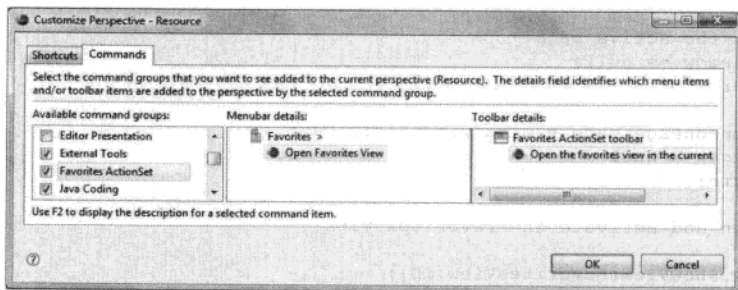


图6-14 自定义透视图对话框

6.6.8 为新操作添加测试

在完成整个任务之前,你需要为新建的Open Favorites View视图设计一个测试。你已经拥有一个可以获取通用测试功能的FavoritesViewTest(参见2.8.3节)。

为所有测试创建一个新的超类,名为AbstractFavoritesTest,然后从已有的FavoritesViewTest获取delay()、assertEquals()和waitForJobs()方法。VIEW_ID常量与FavoritesView.ID常量一致,因此用FavoritesView.ID替换它。然后,创建一个新的继承AbstractFavoritesTest的测试,用于检验新建的OpenFavoritesViewActionDelegate类。

```
package com.qualityeclipse.favorites.test;

import ...

public class OpenFavoritesViewTest extends AbstractFavoritesTest {
    public OpenFavoritesViewTest(String name) {
        super(name);
    }
}
```

覆盖setUp()方法以保证在测试执行前,系统处于合适的状态。

```
protected void setUp() throws Exception {
    super.setUp();

    // Ensure that the view is not open.
    waitForJobs();
    IWorkbenchPage page = PlatformUI.getWorkbench()
        .getActiveWorkbenchWindow().getActivePage();
}
```

```

IViewPart view = page.findView(FavoritesView.ID);
if (view != null)
    page.hideView(view);

// Delay for 3 seconds so that
// the Favorites view can be seen.
waitForJobs();
delay(3000);
}

```

最后，创建测试方法以检验OpenFavoritesViewActionDelegate类。

```

public void testOpenFavoritesView() {

    // Execute the operation.
    (new Action("OpenFavoritesViewTest") {
        public void run() {
            IWorkbenchWindowActionDelegate delegate =
                new OpenFavoritesViewActionDelegate();
            delegate.init(PlatformUI.getWorkbench()
                .getActiveWorkbenchWindow());
            delegate.selectionChanged(this, StructuredSelection.EMPTY);
            delegate.run(this);
        }
    }).run();

    // Test that the operation completed successfully.
    waitForJobs();
    IWorkbenchPage page = PlatformUI.getWorkbench()
        .getActiveWorkbenchWindow().getActivePage();
    assertTrue(page.findView(FavoritesView.ID) != null);
}

```

进入上面的测试后，以下错误将会出现在Problems视图中：

```

Access restriction: The type OpenFavoritesViewActionDelegate
is not accessible due to restriction on required project
com.qualityeclipse.favorites.

```

该内容表示com.qualityeclipse.favorites插件没有为其他插件提供OpenFavorites ViewAction-Delegate的访问方法。为了解决该问题，打开插件清单编辑器至Exported Packages部分（参见2.8.1节），点击Add...，选择com.qualityeclipse.favorites.actions包，并保存更改。

现在，为执行测试的所有项都准备好了。与其独立启动每一个测试，FavoritesViewTest和OpenFavoritesViewTest可以联合成一个名为FavoritesTestSuite的单个测试集。该测试集启动将立即执行这两个测试：

```

package com.qualityeclipse.favorites.test;

import ...

public class FavoritesTestSuite
{
    public static Test suite() {

        TestSuite suite =
            new TestSuite("Favorites test suite");
    }
}

```



```
suite.addTest(  
    new TestSuite(FavoritesViewTest.class));  
  
suite.addTest(  
    new TestSuite(OpenFavoritesViewTest.class));  
  
return suite;  
}  
}
```

虽然当前只有两个测试的情况下，独立启动测试不是一个问题，在将来，当更多测试被添加至Favorites插件后，使用一个单一测试集将节省时间。要启动测试集，在启动菜单中选择Run...，选择在2.8.4节中创建的FavoritesViewTest启动配置文件，并修改目标为新建的FavoritesTestSuite测试集。

6.6.9 讨论

是否定义最高级菜单是一个问题。一方面，最高级菜单可以极大提升刚安装的新产品的性能，它也是让潜在用户熟悉新功能的良好方法；另一方面，如果每一个插件定义一个最高级菜单，那么工具栏将变得很凌乱，而Eclipse将很快变得不可用。此外，如果用户不想看见菜单但总是需要使用1.2.2节中说明的繁杂步骤来移除菜单，那么他将会十分困扰。那该怎么做？

操作集是回答该问题的一个答案。他们可以在plugin.xml中指定以在所有透视图的所有地方可见。使用新建的IActionSetDescriptor.setInitiallyVisible()方法，你可以在程序中覆盖由plugin.xml指定的可见性，这样最高级菜单将不会在任何新打开的透视图图中显示。你可以创建一个新操作以从所有当前和以后的透视图图中移除最高级菜单，通过使用setInitiallyVisible()和IWorkbenchPage.hideActionSet()。你的产品可以在首选项页包含一个单选框（参见12.2节），以使用该操作来显示或隐藏你的最高级菜单。

注意 我们提交了一个Eclipse属性请求和补丁（参见Bugzilla条目#39455于bugs.eclipse.org/bugs/show_bug.cgi?id=39455）以用于这里讨论的IActionSetDescriptor API。这演示了用户如何为Eclipse添加项（参见21.6.4节），以使Eclipse成为一个更好的平台。

另一种选择是将你的最高级菜单或操作集与一个特定的透视图绑定（参见10.2.3节）。在这种方法中，菜单和操作集仅对于特定的活动透视图可用。如果一个或多个透视图十分适合由你插件添加的功能，那么这将是最好的办法。

如果操作是编辑器相关的怎么办？6.9.2节和6.9.5节讨论了添加与特定编辑器相关的菜单和操作。通过使用这种方法，最高级菜单仅当某种特定类型的编辑器打开时才可见。

org.eclipse.ui.actionSetPartAssociations扩展点还提供了另一种选择。它允许无论一个或多个指定类型的视图或编辑器打开时显示操作集，而与它们所在的透视图无关。这是一种很好的保证特定操作出现于多个透视图，而不需要将操作添加至这些透视图的办法。

本章剩下内容关注于提供操作，在视图相关的菜单中，或作为指向特定类型的对象而不是最高级菜单的任务。在这种方法中，操作仅当需要它和它所适用于的特定类型的对象时可见。这种方法防止出现最高级菜单问题，并且可以防止Eclipse变得十分凌乱。用于本地范围操作的不同方法将在剩下小节中进行讨论。

6.7 对象操作

假设你需要让用户较容易地向Favorites添加文件和目录。对象添加项对于该问题是理想的解决

办法。因为仅当当前视图或编辑器的选择包含适用于该操作的对象时，它们才出现于上下文菜单中（图6-15）。在这种方式中，对象添加项当用户需要该操作时可用，而操作不适用时将不会出现。对象操作与具有locationURI id为“org.eclipse.ui.any.popup”的菜单添加项十分类似（参见6.2.9节）。

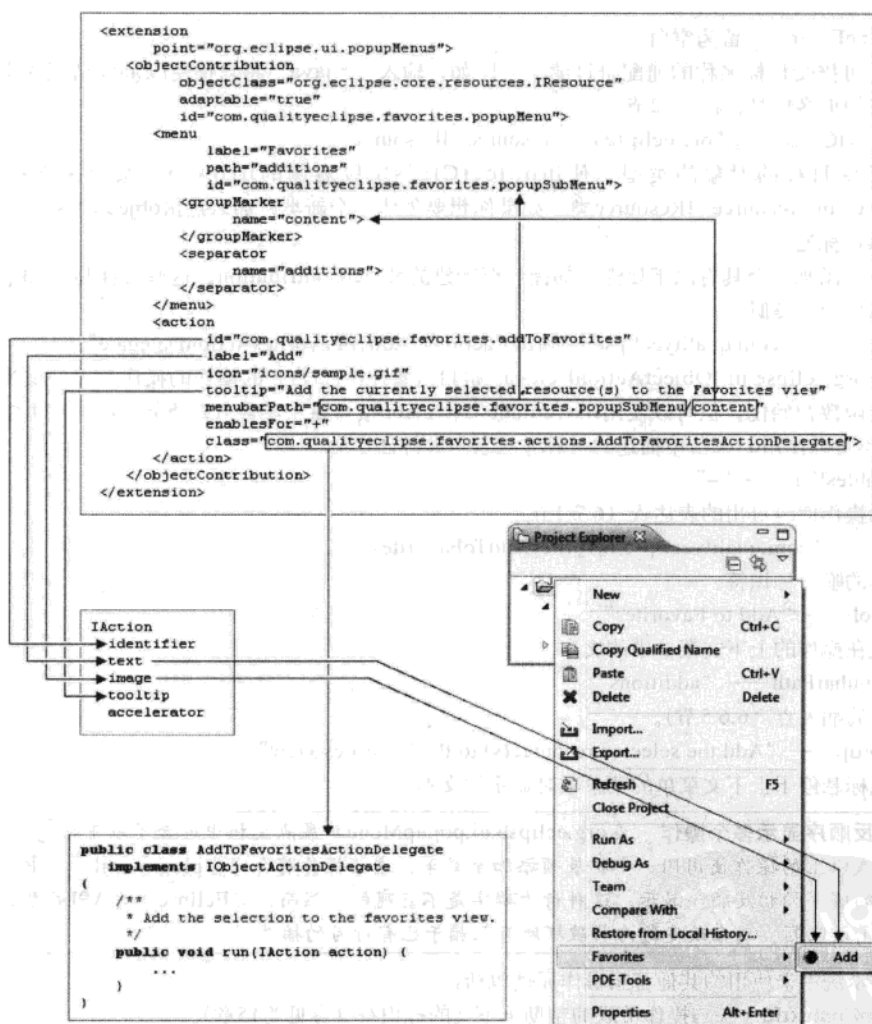


图6-15 对象操作

6.7.1 定义基于对象的操作

如同6.6.1节及其后续几节的内容一样，使用插件清单编辑器的Extensions页面来创建新的对象添加项。点击Add按钮以添加一个org.eclipse.ui.popupMenus扩展项，然后添加一个具有如下属性的objectContribution：

- adaptable——“true”

表示匹配于IResource的对象是可接受的目标（参见21.3节）。

- id——“com.qualityeclipse.favorites.popupMenu”

该添加项的唯一标识符。

- nameFilter——留为空白

指定可接受目标名称的通配符过滤器。比如，输入“*.java”将只接受以.java结尾的文件。关于该内容的更多信息，在6.7.2节。

- objectClass——“org.eclipse.core.resources.IResource”

可接受目标的对象的类型。使用objectClass字段右侧的Browse...按钮来选择已有的org.eclipse.core.resources.IResource类。如果你想要创建一个新类，那么点击objectClass字段左侧的objectClass:标签。

然后，添加一个具有以下属性值的操作至新建的objectContribution。这些属性与6.6.3节中描述的操作属性十分类似。

- class——“com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate”

实现org.eclipse.ui.IObjectActionDelegate接口（参见6.7.3节）的操作的操作代表。该类由它的无参构造函数初始化，但可以使用IExecutableExtension接口赋予参数（21.5节，扩展点中定义的类型）。该类可以使用6.6.6节中描述的三种方法之一进行自定义。

- enablesFor——“+”

表示操作何时可用的表达式（6.7.2节）。

- id——“com.qualityeclipse.favorites.addToFavorites”

操作的唯一标识符。

- label——“Add to Favorites”

出现在操作的上下文菜单中的文本。

- menubarPath——“additions”

操作的插入点（6.6.5节）。

- tooltip——“Add the selected resource(s) to the Favorites view”

当鼠标悬停于上下文菜单的菜单项时显示的文本。

以相反顺序显示多个操作 在org.eclipse.ui.popupMenu扩展点文档中包含了以下信息：“如果两个或以上的操作使用同一个扩展项添加至菜单，这些操作将会以在plugin.xml文件中它们所列出的顺序的相反顺序显示。这种行为确实是不直观的。然而，当Eclipse平台API成型之后才发现该问题。现在改变行为将破坏所有依赖于已有行为的插件。”

在该示例中未使用的其他可用操作属性包括：

- helpContextId——与操作关联的帮助上下文的标识符（参见第15章）。

- icon——相关的图像（参见6.6.4节）。

- overrideActionId——指定该操作所覆盖操作的标识符的可选属性。

- state——对于一个具有单选按钮或开关样式的操作，设置初始状态为true或false（6.6.3节）。

- style——定义操作的可视帧的属性。这项内容在6.6.3节中讨论。而pulldown样式不适用于对象添加项。

6.7.2 操作过滤与可用

为了与惰性载入插件保持一致，Eclipse提供了多种声明机制以基于上下文过滤操作，以及仅当合适时使能可视操作。因为它们是在插件清单中声明的，这些机制具有它们不需要将插件载入Eclipse以使用的优点。

1. 基本过滤和使能 (enablement)

在6.7.1节中，nameFilter和objectClass属性是过滤的示例。而enablesFor属性定义了操作可用的时间。当激活上下文菜单时，如果选择不包含名称与通配符nameFilter匹配的对象，或者不是由objectClass属性所指定的类型，在那种对象添加项中定义的操作将不会出现于上下文菜单中。此外，enablesFor属性使用表6-1中的语法以精确定义要特定操作可用，需要选择多少对象：

该表中列出的技术是那些最常用的用于限制可见性和使能操作的。有些时候，需要一个更好的方法。visibility和filter元素提供了限制操作可见性的另一种方法，而selection和enablement元素提供了一种更灵活的方法以指定操作可用的时机。更高要求的操作使能方法可以在操作代表中使用selectionChanged()方法，如同6.6.6节中描述的那样。

表6-1 enablesFor属性选项

语 法	描 述
!	0个项目被选中
?	0或1个项目被选中
+	1个或1个以上的项目被选中
multiple, 2+	2个或2个以上的项目被选中
n	被选中项目的个数。比如，enablesFor=“4”表示仅当4个项目被选中时才使能该操作
*	任意数量的项目被选中

2. visibility元素

相对于对象添加项的nameFilter和objectClass，visibility元素提供了另一种更强大的替代方法以指定对象添加项的操作对于用户可用的时机。比如，另一种指定用于对象添加项过滤的替代方法将如下所述：

```
<objectContribution ...>
  <visibility>
    <objectClass
      name="org.eclipse.core.resources.IResource"/>
    </visibility>
    ...the other stuff here...
  </objectContribution>
```

如果操作仅对于非只读资源可见，那么visibility对象关系可能和如下内容类似：

```
<objectContribution ...>
  <visibility>
    <and>
      <objectClass
        name="org.eclipse.core.resources.IResource"/>
      <objectState name="readOnly" value="false"/>
    </and>
  </visibility>
  ... the other stuff here ...
</objectContribution>
```

作为<visibility>元素声明的一部分，你可以使用嵌套的<and>、<or>和<not>元素以用于逻辑表达式，再加上以下布尔表达式。

- **adapt**——使被选择的对象适应于指定类型。然后在所有子表达式中使用新建的对象。比如，你想要将选中对象（21.3节）适用于某种资源，然后测试某些资源对象状态。表达式将和以下内容类似：

```
<adapt type="org.eclipse.core.resources.IResource">
  <objectState name="readOnly" value="false"/>
</adapt>
```

adapt表达式的后代是通过使用and操作符相结合。如果适配器或被引用的类型还没有被载入，那么表达式将返回EvaluationResult.NOT_LOADED，如果类型的名称不存在，那么它在验证期内将抛出一个ExpressionException。

- **and**——当所有子元素表达式值都为true的时候，该值为true。
- **instanceof**——将选中对象的类与某个名称相比较。这和objectClass元素是一样的，除了instanceof可以使用and和or元素与其他元素联合。
- **not**——当它的子元素表达式为false时，值为true。
- **objectClass**——与上述内容类似，将选中对象的类与某个名称相比较。
- **objectState**——将选中对象的状态与指定状态相比较。该指定状态与filter元素类似（参见6.7.2节）。
- **or**——当一个子元素表达式值为true时，该值为true。
- **pluginState**——比较插件状态，以显示它是installed还是activated。比如，仅当安装了org.eclipse.pde插件后，类似于<pluginState id="org.eclipse.pde" value="installed"/>的表达式将一个对象添加项设置为可见。而类似于<pluginState id="org.eclipse.pde" value="activated"/>的表达式仅当以某种方式激活了org.eclipse.pde插件时才将对象添加项设置为可见。
- **systemProperty**——比较系统属性。比如，仅当语言为英语时，对象添加项才应当是可见的，那么，该表达式应该为<systemProperty name="user.language" value="en"/>。
- **systemTest**——与systemProperty元素类似。除了systemTest可以使用and和or元素与其他元素结合。
- **test**——设置对象的属性状态。比如，如果一个对象插件仅当Java项目中的一个资源被选中时才可见，那么表达式应为：

```
<test
  property="org.eclipse.debug.ui.projectNature"
  value="org.eclipse.jdt.core.javanature"/>
```

当属性测试程序执行还未被载入的实际测试时，测试表达式返回EvaluationResult.NOT_LOADED。可测试属性集可以使用org.eclipse.core.expressions.propertyTesters扩展点进行扩展。这样的示例之一是org.eclipse.debug.internal.ui.ResourceExtender类。

3. filter元素

filter元素是之前讨论过的objectState元素的简化样式。比如，如果对象添加项对于任意非只读文件可用，那么对象添加项可以和以下内容类似：

```
<objectContribution ...>
  <filter name="readOnly" value="false"/>
  ... the other stuff here ...
</objectContribution>
```

如同objectState元素一样，filter元素使用IActionFilter接口以决定选择中的对象是否与条件相匹配。每个被选中的对象必须实现或适配于IActionFilter接口（第20章有更多关于适配器的内容），并

在testAttribute()方法中实现合适的行为以测试指定的名/值对与指定对象的状态是否匹配。对于资源，Eclipse提供了以下内建状态比较方法，该方法位于org.eclipse.ui.IResourceActionFilter类中：

- name——比较文件名。“*”可以用于开始或结尾以表示“一个或多个字符”。
- extension——比较文件扩展名。
- path——比较文件路径。“*”可以用于开始或结尾以表示“一个或多个字符”。
- readOnly——比较文件的只读属性。
- projectNature——比较项目的自然属性。
- persistentProperty——比较选中资源的持久属性。如果值是一个简单字符串，那么该项测试该资源的属性的存在性。如果它具有“属性名=属性值”的形式，该属性将获取指定名称的属性值并测试它是否等于指定值。
- projectPersistentProperty——比较选中资源所在项目的持久属性。语义和上面列出的persistentProperty类似。
- sessionProperty——比较选中资源的会话属性。语义和上面列出的persistentProperty类似。
- projectSessionProperty——比较选中资源所在项目的会话属性。语义和上面列出的persistentProperty类似。

4. selection元素

selection元素是一项用于使能基于其名称和类型的独立操作的技术。与nameFilter和objectClass属性决定对象添加项中的所有操作是否可见的方式类似。比如，使用selection元素的对象添加项的一种可能的形式是：

```
<objectContribution
    objectClass="java.lang.Object"
    id="com.qualityeclipse.favorites.popupMenu">
    <action
        label="Add to Favorites"
        tooltip="Add the selected resource(s) to the
            Favorites view"
        class="com.qualityeclipse.favorites.actions.
            AddToFavoritesActionDelegate"
        menubarPath="additions"
        enablesFor="+
            id="com.qualityeclipse.favorites.addToFavorites">
        <selection
            class="org.eclipse.core.resources.IResource"
            name="*.java"/>
        </action>
    </objectContribution>
```

使用该声明，对象添加项的操作将总是可见的，但是Add to Favorites操作仅当选择只包含匹配名称过滤“*.java”的IResource的实现时才可用。

5. enablement元素

enablement元素相对于selection元素而言是一种功能更强大的选择。它支持visibility元素所支持的同样复杂的条件逻辑表达式和比较方法（参见6.7.2节）。比如，作为前一节描述的对象添加项声明的具有同样行为的替代方案可能为：

```
<objectContribution
    objectClass="java.lang.Object"
```

```

        id="com.qualityeclipse.favorites.popupMenu">
<action
    label="Add to Favorites"
    tooltip="Add the selected resource(s)
        to the Favorites view"
    class="com.qualityeclipse.favorites.actions.
        AddToFavoritesActionDelegate"
    menubarPath="additions"
    enablesFor="+
    id="com.qualityeclipse.favorites.addToFavorites">
<enablement>
    <and>
        <objectClass
            name="org.eclipse.core.resources.IResource" />
        <objectState name="name" value="*.java" />
    </and>
    </enablement>
</action>
</objectContribution>

```

6. 内容敏感的对象添加项

这对于过滤基于资源内容的操作是一种新的机制。该过滤在插件清单中指定（而不载入你的插件）并通过监视文件内容以决定操作是否应可见或可用。比如，Run Ant...命令与名为build.xml的资源相关联，而不是其他资源。如果你的Ant脚本位于一个叫export.xml的文件中怎么办？这项新机制可以决定Run Ant...命令应基于文件中指定的第一个XML标签还是DTD来可见。在这种情况下，org.eclipse.ant.core插件定义了一个新的antBuildFile内容类型：

```

<extension point="org.eclipse.core.runtime.contentTypes">
    <content-type
        id="antBuildFile"
        name="%antBuildFileContentType.name"
        base-type="org.eclipse.core.runtime.xml"
        file-names="build.xml"
        file-extensions="macrodef,ent,xml"
        priority="normal">
        <describer
            class="org.eclipse.ant.internal.core.
                contentDescriber.AntBuildfileContentDescriber">
        </describer>
    </content-type>
</extension>

```

之前的声明将antBuildFile内容类型与AntBuildfileContentDescriber类关联起来，AntBuildfileContentDescriber类决定XML的内容是否是Ant内容。antBuildFile内容类型可以被用于指定操作可见性和可用性、编辑器关联等等。为了了解更多关于声明和使用你自己的内容类型，参考以下内容：

- 位于eclipse.org > projects > The Eclipse Project > Platform > UI > Development Resources > Content Sensitive Object Contributions的“Content Sensitive Object Contributions”，或浏览dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/object-aware-contributions/objCont.htm。
- 位于Help > Help Contents > Platform Plug-in Developer Guide > Programmer's Guide > Runtime overview > Content types的Eclipse帮助系统中的“Content types”。

- 位于dev.eclipse.org/viewcvs/index.cgi/platform-core-home/documents/ content_ types.html? rev=1.11的“A central content type catalog for Eclipse”。
- 位于eclipse.org/eclipse/platform-core/planning/3.0/plan_content_types.html的“Content types in Eclipse”。

6.7.3 IObjectActionDelegate

让我们回到Favorites插件，下一个任务是创建一个实现IObjectActionDelegate的操作代表。该操作代表在新建的Add to Favorites菜单项的后台执行相关操作。根据下述内容创建一个新的AddToFavoritesActionDelegate类。由于Favorites视图还不是完全运转的，我们将要创建的操作将显示信息而不是添加选中项至视图（参见7.3.1节以了解更多实现细节）。

我们从选择在6.7.1节中定义的操作开始。然后，点击类字段左侧的class:标签。这将打开New Java Class向导以创建一个新的Java类。根据需要填充包和类的名称字段。并且，我们需要确认添加IObjectActionDelegate作为实现的接口，然后点击Finish以生成该类。

然后，添加一个新的字段并修改setActivePart()方法以缓存操作所在的视图或编辑器：

```
private IWorkbenchPart targetPart;

public void setActivePart(IAction action, IWorkbenchPart part) {
    this.targetPart = part;
}
```

最后，修改run()方法以打开一个显示该操作被成功执行的消息对话框。与前面提及的相似，该操作表示将会在7.3.1节中进一步充实。

```
public void run(IAction action) {
    MessageDialog.openInformation(
        targetPart.getSite().getShell(),
        "Add to Favorites",
        "Triggered the " + getClass().getName() + " action");
}
```

6.7.4 创建基于对象的子菜单

菜单可以用于与添加操作类似的方式添加项至上下文菜单。如果三个或以上的类似操作要被添加，那么请考虑将这些操作放置于一个子菜单中而不是上下文菜单本身。Favorites插件仅添加了一个操作至上下文菜单，但是让我们将操作放置于子菜单中而不是上下文菜单本身。

为了创建Favorites菜单，在插件清单编辑器中右键点击com.qualityeclipse.favorites.popupMenu，选择New > menu，为该新建菜单输入以下值：

- id——“com.qualityeclipse.favorites.popupSubMenu”

该子菜单的标识符。

- label——“Favorites”

作为出现在上下文菜单中的子菜单的名称。

- path——“additions”

确定子菜单在上下文菜单中出现位置的插入点（参见6.6.5节）。

然后，将一个groupMarker添加至菜单。该groupMarker的名称为“content”，并包含一个名为“additions”的分隔线（参见6.6.2节）。最后，根据以下内容修改Add to Favorites操作的属性，以使

操作成为新建的Favorites子菜单的一部分：

- label——“Add”

作为操作的名称出现于子菜单的文本。

- menubarPath——“com.qualityeclipse.favorites.popupSubMenu/content”

确定收藏夹子菜单操作出现位置的插入点（参见6.6.5节）。

6.7.5 手动测试新操作

当Favorites Runtime Workbench配置文件启动时（参见2.6节）所有在工作台资源上激活的上下文菜单将包含具有Add子菜单的Favorites菜单。选中该子菜单项将显示一个消息框提醒你操作确实已被正确触发。

仅当根据6.7.2节中的声明中的enablesFor=“+”选中一个或多个对象时，Favorites操作才会显示。这意味着当你测试Favorites菜单时，你必须在运行时工作台具有至少一个项目，并当你激活上下文菜单时，在Navigator视图中选择至少一个资源。如果你没有选中任何项而右键点击，你将只能看到一个不包含Favorites菜单的省略上下文菜单。

6.7.6 为新操作添加测试

最后一项任务是创建一个自动测试。该测试将触发操作并验证结果。因为这项操作显示消息而不是添加资源至Favorites视图。用于验证该测试结果的代码将必须等到下一章（参见7.6节）。我们将会在下一章进一步完善Favorites视图。现在，在Favorites测试项目中创建以下新的测试用例，然后修改Favorites测试包以包含该新的测试（参见6.6.8节）。

你可以首先创建一个继承AbstractFavoritesTest的新的AddToFavoritesTest类，并将它添加至Favorites测试包。

```
package com.qualityeclipse.favorites.test;

import ...

public class AddToFavoritesTest extends AbstractFavoritesTest {

    public AddToFavoritesTest(String name) {
        super(name);
    }
}
```

然后，添加一个字段并覆盖setUp()方法，为测试时期创建一个名为“TestProj”的临时项目。tearDown()方法将在测试完成时删除该临时项目。

要使这些更改可以正确编译，编辑Favorites测试项目的插件清单，并添加org.eclipse.core.resources插件至所需的插件列表（图2-10）。

```
protected IProject project;

protected void setUp() throws Exception {
    super.setUp();
    IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
    project = root.getProject("TestProj");
    project.create(null);
    project.open(null);
}
```

```
protected void tearDown() throws Exception {
    super.tearDown();

    // Wait for a bit for the system to catch up
    // so that the delete operation does not collide
    // with any background tasks.
    delay(3000);
    waitForJobs();

    project.delete(true, true, null);
}
```

最后，添加方法以测试用于添加对象至Favorites视图的新的菜单项。

```
public void testAddToFavorites() throws CoreException {

    // Show the resource navigator and select the project.
    IViewPart navigator = PlatformUI.getWorkbench()
        .getActiveWorkbenchWindow().getActivePage().showView(
        "org.eclipse.ui.views.ResourceNavigator");
    StructuredSelection selection = new StructuredSelection(project);
    ((ISetSelectionTarget) navigator).selectReveal(selection);

    // Execute the action.
    final IObjectActionDelegate delegate
        = new AddToFavoritesActionDelegate();
    IAction action = new Action("Test Add to Favorites") {
        public void run() {
            delegate.run(this);
        }
    };
    delegate.setActivePart(action, navigator);
    delegate.selectionChanged(action, selection);
    action.run();

    // Add code here at a later time to verify that the
    // Add to Favorites action correctly added the
    // appropriate values to the Favorites view.
}
```

6.8 视图操作

有几种方法可以使操作作为视图的一部分列入清单。比如，Members视图（Java Browsing透视图的一部分；参见图1-5）具有在它的标题栏出现的工具栏按钮，出现在工具栏按钮右侧的下拉菜单，和包含更多操作的上下文菜单（图6-16）。操作使用扩展点机制添加至视图，与前面两节讨论的类似。此外，视图可以在程序中提供它们自身的操作，绕过扩展点机制（参见7.3节）。

6.8.1 定义视图上下文子菜单

与objectContribution类似，viewerContribution用于添加菜单项至上下文菜单。而objectContribution使菜单项根据查看器中的选择来显示，viewerContribution使菜单项根据查看



图6-16 视图操作

器的类型来显示。与objectContribution类似，viewerContribution元素可以具有单个的visibility子元素。该子元素当viewerContribution元素的所有其他子元素对于用户可见时，取得控制权（参见6.7.2节）。

Favorites子菜单在几种不同类型的视图中显示，但不在Members视图中显示。可能使用6.7节中描述的objectContribution实现方法对于以Members视图中包含的对象为目标是更合适的。然而，我们使用viewerContribution作为示例。

我们从右键点击popupMenu扩展项开始。该扩展项被添加至6.7.1节的一部分。然后，选择New > viewerContribution。为新添加的viewerContribution添加下列属性。

- id——“com.qualityeclipse.favorites.membersViewPopup”

该视图添加项的标识符。

- targetID——“org.eclipse.jdt.ui.MembersView”

子菜单将要添加至的视图上下文菜单的标识符（参见21.6节）。

通过右键点击viewerContribution并选择New > menu，可以将Favorites菜单添加至Members视图上下文菜单。为新的菜单输入以下属性：

- id——“com.qualityeclipse.favorites.membersViewPopupMenu”

Members视图上下文菜单中的Favorites菜单的标识符。

- label——“Favorites”

作为Favorites子菜单的名称出现于Members视图上下文菜单的文本。

- path——“additions”

决定Favorites子菜单出现在Members视图上下文菜单中的位置的插入点（参见6.6.5节）。

然后，将一个名为“content”并具有一个名为“additions”的分隔线的groupMarker添加至菜单（参见6.6.2节）。

6.8.2 定义视图上下文菜单操作

最后，右键点击viewerContribution，选择New > action以添加一个操作至Favorites子菜单。并为新操作输入以下属性：

- class——“com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate”

实现org.eclipse.ui.IViewActionDelegate接口并执行该操作的类的完全合格名称。在这种情况下，在对象添加项中使用的同一个操作代表也在这里使用，但做了一些更改（参见6.8.3节）。该类使用它的无参数构造函数进行初始化，但可以使用IExecutableExtension接口指定参数（参见21.5节）。

- id——“com.qualityeclipse.favorites.addToFavoritesInMembersView”

该操作的标识符。

- label——“Add”

出现在Favorites子菜单中的操作的名称。

- menubarPath——“com.qualityeclipse.favorites.membersViewPopupMenu/content”

指定操作将出现在Favorites子菜单中的位置的插入点（参见6.6.5节）。如果操作将直接出现在Members视图上下文菜单而不是Favorites子菜单中，使用值“additions”作为替换。

- tooltip——“Add selected member's compilation unit to the Favorites view”

描述操作的文本。

其他可用的操作属性但在这里没有使用的包括以下内容。

- `enablesFor`——表示操作将可用的时机的表达式（参见6.7.2节）。
- `helpContextId`——与操作关联的帮助上下文的标识符（参见第15章）。
- `icon`——相关图像（参见6.6.4节）。
- `overrideActionId`——指定该操作覆盖的操作的标识符的可选属性。
- `state`——对于具有单选按钮或开关样式的操作，设置初始状态为`true`或`false`（参见6.6.3节）。
- `style`——定义操作可视帧的属性。这项内容在6.6.3节中提到，而例外的是，`pulldown`样式不适用于对象添加项。

与6.7.2节中类似，你还可以为操作元素指定`selection`和`enablement`子元素。

6.8.3 IViewActionDelegate

视图添加项的操作代表必须实现`org.eclipse.ui.IViewActionDelegate`接口。所以你首先需要修改在6.7.3节中介绍的`AddToFavoritesActionDelegate`类。首先，添加`IViewActionDelegate`接口至实现语句，然后添加以下`init()`方法以缓存目标部分。该操作代表的其他所有部分保持不变。

```
public void init(IViewPart view) {  
    this.targetPart = view;  
}
```

6.8.4 定义视图工具栏操作

除了位于视图上下文菜单中的`Favorites`子菜单之外，操作还应当作为一个工具栏按钮出现于`Members`视图（参见7.3.3节以在程序中添加工具栏按钮至视图）中。与6.6.1节和后续小节中描述的类似，使用插件清单编辑器的`Extensions`页来创建新的视图添加项。点击`Add`按钮以添加一个`org.eclipse.ui.viewActions`扩展项，然后向该扩展项中添加一个具有以下属性的`viewContribution`。

- `id`——“`com.qualityeclipse.favorites.membersViewActions`”

视图添加项的标识符。

- `targeted`——“`org.eclipse.jdt.ui.MembersView`”

操作被添加至的视图的标识符。

然后，通过右键点击`viewContribution`并选择`New > action`来向`Members`视图工具栏添加操作，然后为新操作输入下面列出的属性。所有在6.7.1节中列出的`objectContribution`操作属性，也适用于`viewContribution`操作。

- `class`——“`com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate`”

实现`org.eclipse.ui.IViewActionDelegate`接口并执行操作的类的完全合格名称。在这种情况下，在对象添加项中使用的同一个操作代表也在这里用到，但经过了一些修改（参见6.8.3节）。

- `icon`——“`icons/sample.gif`”

显示于视图工具栏的操作的图标。

- `id`——“`com.qualityeclipse.favorites.addToFavoritesInMembersView`”

操作的标识符。

- `toolbarPath`——“`additions`”

确定操作将出现于`Members`视图的工具栏的位置的插入点（参见6.6.5节）。

- `tooltip`——“`Add the selected items in the Members view to the Favorites view`”

当指针悬停于与操作相关联的工具栏按钮时，出现提示中的描述操作的文本。

6.8.5 定义视图下拉子菜单和操作

在前面小节中描述的viewContribution扩展项被用于添加视图下拉子菜单（参考7.3.2节以在程序中创建视图下拉菜单）。一般地，视图下拉菜单包括操作，如对于该视图的特定的分类和过滤。要添加Favorites子菜单和操作至Members视图下拉菜单（不仅是在它真正需要的该地方，而是它被添加至的其他所有地方），右键点击viewContribution扩展项，选择New > menu，然后根据以下内容设置新创建的菜单的属性：

- id——“com.qualityeclipse.favorites.membersViewPullDownSubMenu”

Members视图中的Favorites菜单的标识符。

- label——“Favorites”

作为Favorites子菜单的名称出现于Members视图下拉菜单的文本。

- path——“additions”

确定Favorites子菜单将出现于Members视图下拉菜单的位置的插入点（参见6.6.5节）。

然后，添加一个具有名称为“content”和一个名为“additions”的分隔线的groupMarker至该菜单。最后，可以修改在6.8.4节中定义的操作以在菜单中定义菜单项。该菜单与工具栏按钮一同创建。该菜单已经由修改一些它的属性时进行了描述。

- label——“Add”

出现于Favorites子菜单中的操作的名称。

- menubarPath——“com.qualityeclipse.favorites.membersViewPullDownSubMenu/content”

确定操作将出现于Favorites子菜单中的位置的插入点（参见6.6.5节）。如果操作直接出现于Members视图下拉菜单而不是Favorites子菜单中，你需要使用一个“additions”作为替代。

6.8.6 手动测试新操作

当对于插件清单和操作代表的更改已经完成时，启动Runtime Workbench并监视Members视图将显示新的Favorites子菜单和Add to Favorites工具栏按钮。

6.8.7 为新操作添加测试

除了在6.7.6节中创建的测试用例外，不需要任何其他测试用例。这是因为重用了相同的操作代表。在第7章进一步丰富Favorites视图后，可以添加更多用于新类型的选择的测试。

6.8.8 视图上下文菜单标识符

以下是一些Eclipse视图中的上下文菜单标识符的相关内容。为了了解更多关于该列表是如何生成的信息，参考21.6节。

Ant

```
id = org.eclipse.ant.ui.views.AntView  
menuId = org.eclipse.ant.ui.views.AntView
```

Bookmarks

```
id = org.eclipse.ui.views.BookmarkView  
menuId = org.eclipse.ui.views.BookmarkView
```

Breakpoints

```
id = org.eclipse.debug.ui.BreakpointView  
menuId = org.eclipse.debug.ui.BreakpointView
```

Console

```
id = org.eclipse.ui.console.ConsoleView  
menuId = org.eclipse.ui.console.ConsoleView
```

Debug

```
id = org.eclipse.debug.ui.DebugView  
menuId = org.eclipse.debug.ui.DebugView
```

Display

```
id = org.eclipse.jdt.debug.ui.DisplayView  
menuId = org.eclipse.jdt.debug.ui.DisplayView
```

Expressions

```
id = org.eclipse.debug.ui.ExpressionView  
menuId = org.eclipse.debug.ui.VariableView.detail  
menuId = org.eclipse.debug.ui.ExpressionView
```

Members

```
id = org.eclipse.jdt.ui.MembersView  
menuId = org.eclipse.jdt.ui.MembersView
```

Memory

```
id = org.eclipse.debug.ui.MemoryView  
menuId = org.eclipse.debug.ui.MemoryView.MemoryBlocksTreeViewPane
```

Navigator

```
id = org.eclipse.ui.views.ResourceNavigator  
menuId = org.eclipse.ui.views.ResourceNavigator
```

Package Explorer

```
id = org.eclipse.jdt.ui.PackageExplorer  
menuId = org.eclipse.jdt.ui.PackageExplorer
```

Packages

```
id = org.eclipse.jdt.ui.PackagesView  
menuId = org.eclipse.jdt.ui.PackagesView
```

Problems

```
id = org.eclipse.ui.views.ProblemView  
menuId = org.eclipse.ui.views.ProblemView
```

Projects

```
id = org.eclipse.jdt.ui.ProjectsView  
menuId = org.eclipse.jdt.ui.ProjectsView
```

Registers

```
id = org.eclipse.debug.ui.RegisterView  
menuId = org.eclipse.debug.ui.VariableView.detail  
menuId = org.eclipse.debug.ui.RegisterView
```

Tasks

```
id = org.eclipse.ui.views.TaskList
```



```
menuId = org.eclipse.ui.views.TaskList
```

Threads and Monitors

```
id = org.eclipse.jdt.debug.ui.MonitorsView
menuId = org.eclipse.jdt.debug.ui.MonitorsView
```

Types

```
id = org.eclipse.jdt.ui.TypesView
menuId = org.eclipse.jdt.ui.TypesView
```

Variables

```
id = org.eclipse.debug.ui.VariableView
menuId = org.eclipse.debug.ui.VariableView.detail
menuId = org.eclipse.debug.ui.VariableView
```

6.9 编辑器操作

操作可以以一种与它们如何被添加至视图类似的方式添加至编辑器。比如，Java编辑器具有一个上下文菜单。因此，自然地，Favorites操作应出现在这里而不管是否真正需要它（图6-17）。此外，编辑器可以通过使用标准扩展点机制添加操作至它们自身。一些相关的章节包括：

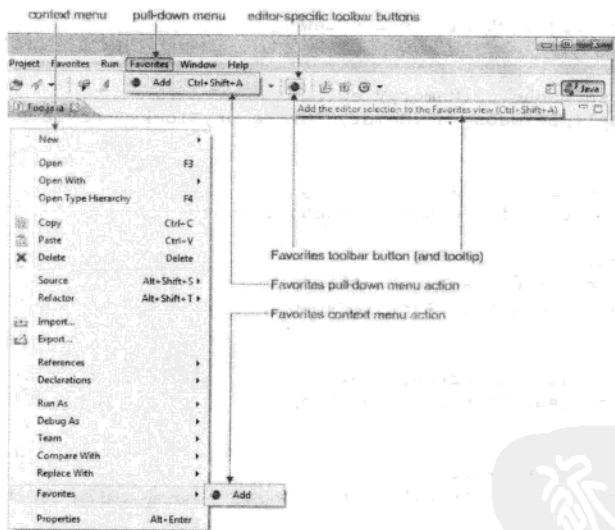


图6-17 编辑器操作

- 8.5节，更多关于编辑器操作的内容。
- 14.2.4节，一个关于操作编辑器中的文本的示例。
- 第17章，更多关于扩展点的内容。

6.9.1 定义编辑器上下文菜单

为了添加Favorites菜单至Java编辑器的上下文菜单，重新回到6.7.1节中声明的popupMenus扩展

项。右键点击它并选择New > viewerContribution。为新建的查看器添加项输入以下属性。与对象添加项类似，visibility子元素可以用于控制菜单和操作出现于编辑器上下文菜单的时机（参见6.7.2节）。

- id——“com.qualityeclipse.favorites.compilationUnitEditorPopup”

查看器添加项的标识符。

- targetID——“#CompilationUnitEditorContext”

操作将要被添加至的编辑器上下文菜单的标识符（参见21.6节以寻找组件标识符）。

然后，右键点击新的查看器添加项扩展项并选择New > menu以在编辑器上下文菜单中创建Favorites子菜单。为新的菜单声明输入以下属性。

- id——“com.qualityeclipse.favorites.compilationUnitEditorPopup SubMenu”

编辑器上下文菜单中的Favorites菜单的标识符。

- label——“Favorites”

作为Favorites子菜单的名称出现于编辑器上下文菜单中的文本。

- path——“additions”

确定Favorites子菜单将出现于编辑器上下文菜单的位置的插入点（参见6.6.5节）。

最后，向菜单添加一个名为“content”并具有一个名为“additions”的分隔线的groupMarker（参见6.6.2节）。

6.9.2 定义编辑器上下文操作

通过右键点击6.9.1节中定义的查看器添加项并选择New > action以添加Add to Favorites操作至Favorites子菜单。输入以下操作属性。

- class——“com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate”

实现org.eclipse.ui.IEditorActionDelegate接口并执行操作的类的完全合格名称。在对象添加项中使用的同一个操作代表在这里也使用，但做出了一些修改（参见6.7.3节）。该类使用它的无参数构造函数进行初始化，但还可以通过使用IExecutableExtension接口赋予参数（参见21.5节）。

- id——“com.qualityeclipse.favorites.addToFavoritesInCompilationUnitEditor”

操作的标识符。

- label——“Add”

出现于Favorites子菜单的操作的名称。

- menubarPath——“com.qualityeclipse.favorites.compilationUnitEditorPopupSubMenu/content”

确定操作将出现于Favorites子菜单中的位置的插入点（参见6.6.5节）。要使操作直接出现于编辑器的上下文菜单而不是Favorites子菜单中，使用一个“additions”作为替代。

其他没有在这里列出的操作属性与6.8.2节中列出的查看器添加项一致。

6.9.3 IEditorActionDelegate

编辑器添加项的操作代表必须实现org.eclipse.ui.IEditorActionDelegate接口，所以你必须修改在6.7.3节初次介绍的AddToFavoritesActionDelegate类。

首先添加IEditorActionDelegate接口至实现语句，然后添加以下setActiveEditor()方法以缓存目标部分。操作代表的所有其他部分保持不变。


```
public void setActiveEditor(IAction action, IEditorPart editor) {  
    this.targetPart = editor;  
}
```

6.9.4 定义编辑器最高级菜单

使用org.eclipse.ui.editorActions扩展点，你可以定义一个工作台窗口菜单和工具栏按钮。它们仅当打开特定类型的编辑器时才可见。和6.6.9节中讨论的那样，在添加菜单或工具栏按钮至工作台窗口本身之前请认真考虑一下。Favorites示例并不真正需要这项内容，但以下内容带领你浏览整个过程，以作为相关内容。

我们首先在插件清单编辑器中的Extension页面中点击Add按钮，并添加一个新的org.eclipse.ui.editorActions扩展项。右键点击新的扩展项并选择New > editorContribution，然后输入以下editorContribution属性。

- id——“com.qualityeclipse.favorites.compilationUnitEditorActions”

编辑器添加项的标识符。

- targetID——“org.eclipse.jdt.ui.CompilationUnitEditor”

应当为这些菜单和操作打开以可见的编辑器的类型标识符。

通过右键点击editorContribution并选择New > menu，添加Favorites菜单。为新菜单输入以下属性。

- id——“com.qualityeclipse.favorites.compilationUnitEditorPopupMenu”

Favorites菜单的标识符。

- label——“Favorites”

作为Favorites子菜单出现于工作台窗口菜单栏的文本。

- path——“additions”

确定Favorites子菜单将出现于工作台窗口菜单栏的位置的插入点（参见6.6.5节）。

最后，向菜单添加一个名为“content”并具有一个名为“additions”的分隔线的groupMarker（参见6.6.2节）。

6.9.5 定义编辑器最高级操作

将操作添加至Favorites菜单包括以下内容：右键点击editorContribution，选择New > action为新操作输入以下属性。与对象添加项类似，selection和enablement元素可以用于限制可见性和使能操作（参见6.7.2节）。

- class——“com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate”

实现org.eclipse.ui.IEditorActionDelegate接口并执行操作的类的完全合格名称。在这种情况下，对象添加项中使用的操作代表在6.9.3节中被修改，因此也可以在这里使用。

- id——“com.qualityeclipse.favorites.addToFavoritesInCompilationUnitEditor”

操作的标识符。

- label——“Add”

用于出现于Favorites菜单中的操作的文本。

- menubarPath——“com.qualityeclipse.favorites.compilationUnitEditorPopupMenu/content”

表示菜单将出现于菜单栏的位置的插入点（参见6.6.5节）。

其他在该示例没有使用但可用的操作属性包括：

- **definitionId**——操作的命令标识符，允许键位序列和操作相关联。为了了解更多细节，参见6.11节。
- **enablesFor**——表示操作何时可用的表达式（参见6.7.2节）。如果该属性内容为空，那么操作将总是可用，除非使用IAction接口在程序中覆盖。
- **helpContextId**——与操作关联的帮助上下文的标识符（参见15.3.1节）。
- **hoverIcon**——当鼠标悬停于操作上但不点击时显示的图像（参见6.6.4节以了解更多细节）。
- **icon**——相关图像（参见6.6.4节）。
- **state**——对于具有单选或开关样式的操作，设置初始状态为true或false（参见6.6.3节）。
- **style**——定义操作的可视帧的属性。具有以下值其中之一：
 - **push**——常规菜单或工具栏项（默认样式）。
 - **radio**——单选按钮样式的菜单或工具栏项。其中每次只有一个项可用（参见state属性）。
 - **toggle**——选中的菜单项或开关工具项（参见state属性）。
- **toolbarPath**——表示按钮出现于工具栏位置的插入点（参见6.6.5节以了解更多细节）。
- **tooltip**——当鼠标悬停在工作台工具栏的操作图标上时显示的文本。

6.9.6 定义编辑器工具栏操作

与工作台菜单操作作为工具栏按钮显示类似，6.9.5节中定义的编辑器操作可以通过对其做出以下属性修改以显示于工作台窗口工具栏：

- **icon**——“icons/sample.gif”
- **toolbarPath**——“Normal/additions”
- **tooltip**——“Add the editor selection to the Favorites view”

6.9.7 为新操作添加测试

和6.8.7节中描述的类似，将会在第7章中为新的选择类型添加与6.7.6节中列出的测试用例相同的测试用例。

6.9.8 编辑器上下文菜单标识符

以下内容是用于一些Eclipse编辑器的上下文菜单标识符。为了了解更多关于该列表是如何生成信息的，参见21.6节。

```
Ant Editor (build.xml)
id = org.eclipse.ant.ui.internal.editor.AntEditor
menuId = #TextEditorContext
menuId = #TextRulerContext

Class File Editor (*.class)
id = org.eclipse.jdt.ui.ClassFileEditor
menuId = #ClassFileEditorContext
menuId = #ClassFileRulerContext

Compilation Unit Editor (*.java)
id = org.eclipse.jdt.ui.CompilationUnitEditor
menuId = #CompilationUnitEditorContext
menuId = #CompilationUnitRulerContext
```



Default Text Editor

```
id = org.eclipse.ui.DefaultTextEditor
menuId = #TextEditorContext
menuId = #TextRulerContext
```

Snippet Editor (*.jpage)

```
id = org.eclipse.jdt.debug.ui.SnippetEditor
menuId = #JavaSnippetEditorContext
menuId = #JavaSnippetRulerContext
```

6.10 操作和键绑定

工作台操作和编辑器操作都可以拥有相关的快捷键（参见7.3.5节以了解如何在程序中关联快捷键）。一开始，快捷键是作为操作声明的一部分进行指定的。但这种办法无法防止多个操作声明同样的快捷键，并且不允许用户更改键绑定。新的方法包括将键绑定（参见6.4节）与操作相关联（图6-18）。

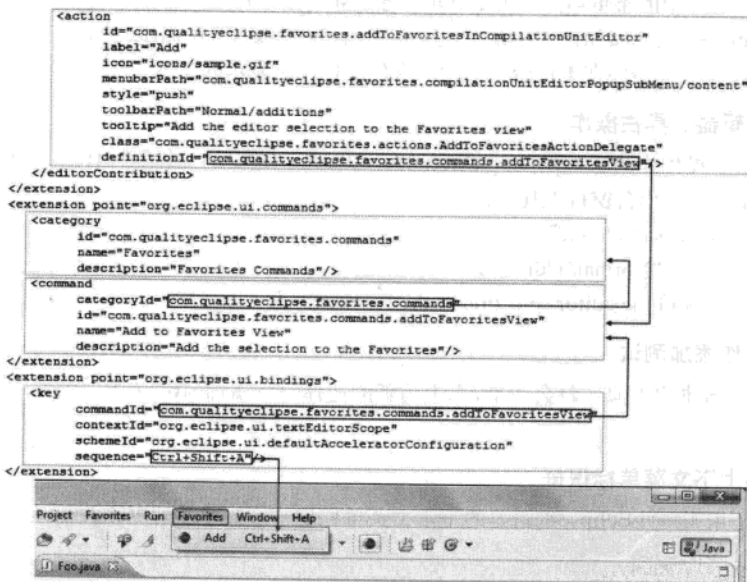


图6-18 键绑定声明

6.10.1 将命令与操作相关联

为Favorites示例定义快捷键包括：

- 创建新的命令（参见6.1节）
- 创建新的键绑定（参见6.4节）
- 修改在6.9.5节中定义的编辑器操作以引用命令

当完成了前两步之后，在插件清单中选择编辑器操作，并修改definitionId属性的值为“com.qualityeclipse.favorites.commands.add”，以使操作现在可以引用命令和相关联的键绑定。

6.10.2 键盘可访问性

键盘可以用来选择工作台窗口中的菜单项。比如，如果你按下并释放Alt键然后按下并释放F键（或按下Alt+F），你将会看到工作台窗口的File菜单被展开。如果你更仔细地观察，你将看到每一个菜单标签和菜单项标签中至少有一个字母具有下划线。当你在这种菜单选择模式中，按下具有下划线的字符将激活该菜单或菜单命令。在一些平台中，比如Windows XP，这些下划线仅当你激活菜单选择模式时才可见。

在你的插件清单中，可以通过为某个字符添加“&”以指定菜单或菜单项的标签中的该字符具有下划线。比如，在以下声明中，单词“Favorites”中的字母“r”前面的“&”使得该字母在你激活菜单选择模式时具有下划线（图6-19）。

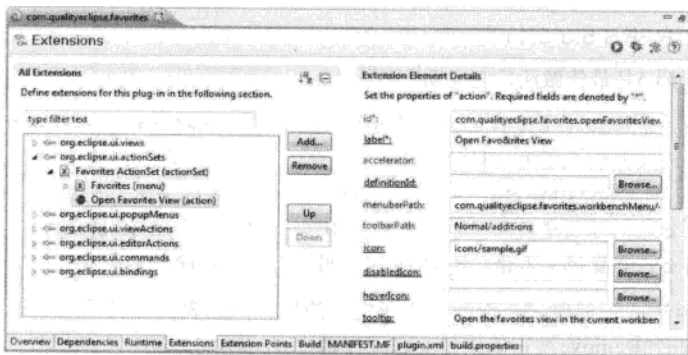


图6-19 显示用于键盘快捷键的“&”的插件清单编辑器

当查看该同一个声明的XML文档时，&字符显示为&这是因为在XML文档中，&字符具有特殊含义。

```
<action
  class="com.qualityeclipse.....OpenFavoritesViewActionDelegate"
  icon="icons/sample.gif"
  id="com.qualityeclipse.favorites.openFavoritesView"
  label="Open Favo&amp;avorites View"
  menubarPath="com.qualityeclipse.favorites.workbenchMenu/content"
  style="push"
  toolbarPath="Normal/additions"
  tooltip="Open the Favorites view in the current workbench page"/>
```

如果你对Favorites菜单声明也使用这种方法（参见6.6.1节），你可以使用一个键击序列以不需要鼠标来打开Favorites视图。

- 按下并释放Alt键以进入菜单选择模式。
 - 按下并释放“v”键以展开Favorites菜单。
 - 按下并释放“r”键以激活Open Favorites View操作。
- 或
- 输入Alt+V以展开Favorites菜单。
 - 按下并释放“r”以激活Open Favorites View操作。

适用于Rational软件 从这一章开始，我们将列出IBM的相关RFRS认证要求并简要讨论通过各项测试所需要做的工作。我们还将努力保证我们正使用的Favorites示例符合所有相关要求。规则定义引用于来源于IBM官方的《Ready for IBM Rational Software Integration Requirements》的许可。为了获取更多关于RFRS项目的信息，参见附录B，或访问IBM网站www.developer.ibm.com/isv/rational/readyfor.html。

6.11 RFRS相关事项

《RFRS Requirements》中的“用户界面”章节包含了处理操作的最佳做法。它来源于Eclipse UI 准则。

全局操作标签 (RFRS 5.3.5.1)

用户界面准则#3.3是最佳做法，它说明：

为新建、删除、添加和移除操作采用工作台标签术语。为保持一致，任何具有和工作台中已有操作类似行为的操作需要采用同样的术语。当创建资源时，术语“new”应被用于操作或向导。比如，“New File”，“New Project”，和“New Java Class”。术语“Delete”应被用于删除一个已有的资源。当在资源内容创建对象时（比如，XML文档中的一个标签），应使用术语“Add”；用户添加一些项至已有资源。术语“Remove”应用于从资源移除对象。

为了通过这项测试，创建一个你的程序所定义的操作的列表，并说明它们的用途。说明术语New、Delete、Add和Remove被正确使用并在工作台中保持一致。在本章稍早时展示的示例所说明的情况下，最好是显示Favorites编辑器操作（图6-17）并向读者描述它们的用途。

6.12 总结

Eclipse用户可以通过使用工作台下拉菜单和工具栏或使用由不同视图定义的上下文菜单来触发命令。这其中的每一个都是操作示例。本章讨论了如何创建不同的操作和如何通过使用过滤器控制它们的可见性和可用状态。

参考文献

本书资源(2.9节)。

Menus Extension Mapping http://wiki.eclipse.org/Menus_Extension_Mapping.

Menu Contributions http://wiki.eclipse.org/Menu_Contributions.

Command Core Expressions http://wiki.eclipse.org/Command_Core_Expressions.

Platform Command Framework http://wiki.eclipse.org/Platform_Command_Framework.

Configuring and adding menu items <https://www.ibm.com/developerworks/library/os-eclipse-3.3menu>.

ScreenCast: Using Property Testers in the Eclipse Command Framework <http://konigsberg.blogspot.com/2008/06/screenCast-using-property-testers-in.html>.

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer's Guide to Eclipse, Second Edition*. Addison-Wesley, Boston, 2004.

第7章 视图

许多插件将添加新的Eclipse视图或改进已有的视图作为提供信息给用户的一种方法。本章包含了以下内容：创建新视图，修改视图以响应活动编辑器或其他视图中的选择，以及将视图选择输出至Eclipse其他部分。此外，本章还简要说明了编辑器和视图之间的区别，以及何时在两者间做出选择。

视图必须实现org.eclipse.ui.IViewPart接口。一般地，视图是org.eclipse.ui.part.ViewPart的子类，因此也间接地成为org.eclipse.ui.part.WorkbenchPart的子类。视图继承了大量用于实现IViewPart接口的行为（图7-1）。

视图包含于视图站点（view site）（org.eclipse.ui.IViewSite类型的实例）中。因此它也包含于工作台页面（workbench page）（org.eclipse.ui.IWorkbenchPage的实例）中。根据延迟初始化的特性，IWorkbenchPage实例包含org.eclipse.ui.IViewReference的实例而不是视图本身，使得视图可以不需要真正载入定义该视图的插件而被列举和引用。

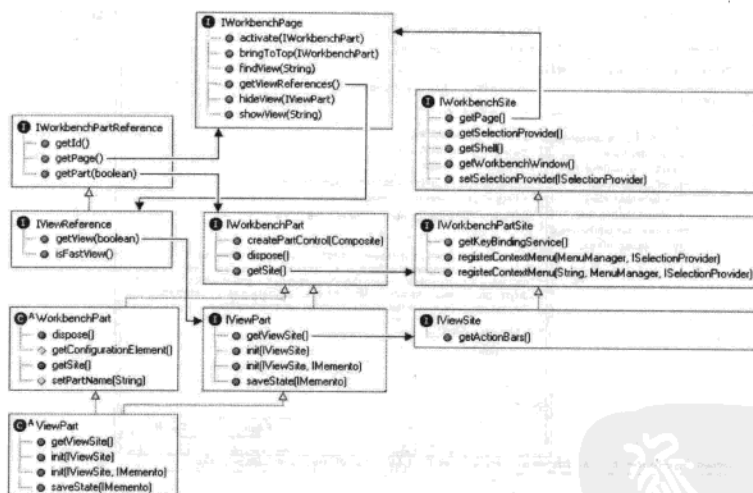


图7-1 ViewPart类

视图通过超类org.eclipse.ui.part.WorkbenchPart和org.eclipse.ui.IWorkbenchPart接口与编辑器共享通用行为集。但它们之间有一些非常重要的区别。任何在视图中执行的操作将立即影响工作区和底层资源的状态。但是编辑器遵循经典的打开-修改-保存样式。

编辑器出现于Eclipse的一个区域，然而视图被安排在编辑器区域的外围（参见1.2.1节）。编辑器一般是基于资源的，而视图可以显示关于一个资源、多个资源，甚至一些与资源完全无关的事物的相关信息。

由于在工作台中可能存在数百个视图，所以它们被分了类。Show View对话框表示一个分组的视图的列表（参见2.5节），以使用户可以更容易地找到想要的视图。

7.1 视图声明

创建新视图包含了三个步骤：

- 在插件清单中定义视图的类别。
- 在插件清单中定义视图。
- 创建包含代码的视图部分。

立即开始着手这些任务的一种方法是当创建插件时就创建视图（参见2.2.3节）。如果插件已经存在，那这将成为一个三个步骤的过程。

7.1.1 声明视图类别

首先，为了定义新的视图类别，编辑插件清单并浏览至Extensions页面。如果没有显示org.eclipse.ui.views扩展项，点击Add...按钮以添加它（图7-2）。当没有新的类别时，右键点击org.eclipse.ui.views扩展项并选择New > category以添加一个新类别。

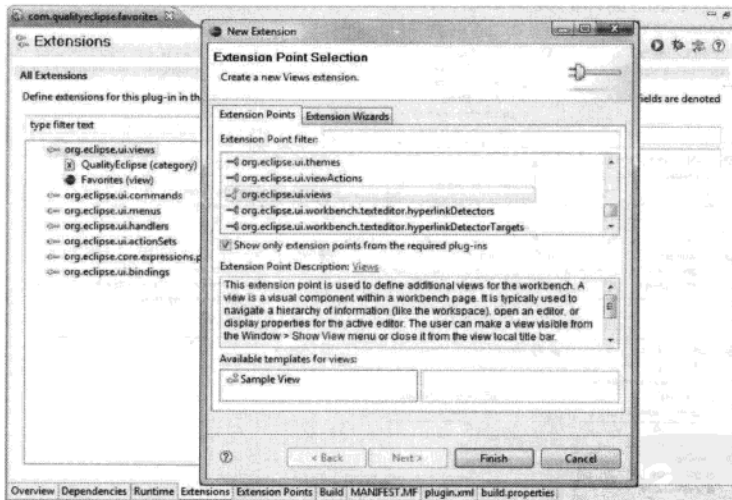


图7-2 选中org.eclipse.ui.views扩展点的新建扩展项向导

该类别的属性可以在插件清单编辑器中修改（图7-3）。对于包含Favorites视图的类别来说，属性应和以下内容一致：

- id——“com.qualityeclipse.favorites”
类别的唯一标识符。
- name——“QualityEclipse”

出现于Show View对话框中的类别的可读名称（图2-20）。

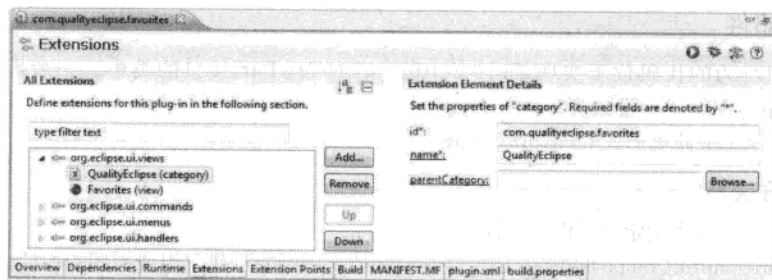


图7-3 显示Quality Eclipse视图类别的插件清单编辑器

7.1.2 声明视图

当定义了视图类别后，右键点击Extensions页的org.eclipse.ui.views扩展项，并选择New > view以定义一个新的视图。使用编辑器的Extension Element Details部分（图7-4）以修改该视图的属性。收藏夹的属性应和以下内容一致：

- category——“com.qualityeclipse.favorites”
包含该视图的视图类别的唯一标识符。
- class——“com.qualityeclipse.favorites.views.FavoritesView”

定义视图并实现了org.eclipse.ui.IViewPart接口的类的完全合格名称。该类使用它的无参构造函数进行初始化，但也可以使用IExecutableExtension接口赋予参数（参见21.5节）。

- icon——“icons/sample.gif”

在Show View对话框和视图的左上角显示的图像（图2-20）。与操作图像（参见6.6.4节）类似，该路径是相对于插件安装目录的路径。

- id——“com.qualityeclipse.favorites.views.FavoritesView”
该视图的唯一标识符。
- name——“Favorites”

显示于视图的标题栏和Show View对话框中的视图的可读名称（图2-20）。

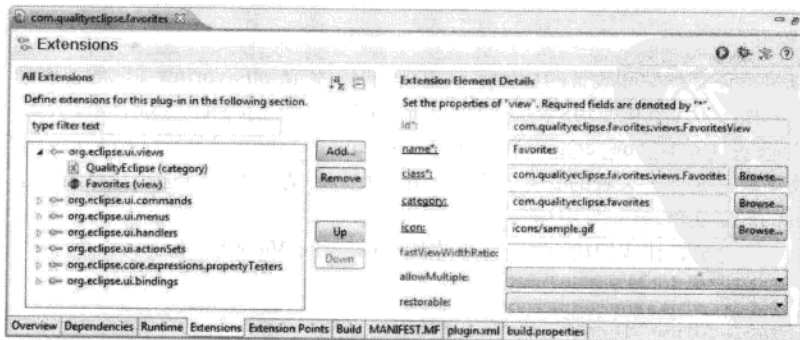


图7-4 显示收藏夹视图的插件清单编辑器

7.2 视图部件

定义视图行为的代码位于实现了org.eclipse.ui.IViewPart接口的类中。一般地，该类继承org.eclipse.ui.part.ViewPart抽象类。

2.3.3节讨论了最简单形式的Favorites视图。

7.2.1 视图方法

IViewPart和它超类型定义了以下方法。

- createPartControl(Composite)——该方法是必需的。因为它创建了组成视图的控件。一般地，该方法可以较容易地调用细粒度的方法，如createTable、createSortActions、createFilters等（参见下一节）。
- dispose()——清理所有系统资源，比如图像、剪贴板等，所有我们在该类中创建的资源。这也遵循Eclipse的整体主题之一：谁创建，谁销毁。
- getAdapter(Class)——返回与特定接口关联的适配器，以使视图可以参与不同的工作台行为（参见后续内容和21.3节）。
- saveState(IMemento)——保存该视图的本地状态，如当前选择、当前排序、当前过滤器等（参见7.5.1节）。
- setFocus()——该方法是必需的。因为它将焦点设置于视图内合适的控件（参见下一节）。

视图可以参与不同的工作台行为，通过直接实现或包含返回特定接口的实例的getAdapter(Class)方法。这些接口的一部分列出如下：

- IContextProvider——动态上下文提供者，用于提供根据不同平台状态而改变的聚焦动态帮助（参见15.3.4节以了解更多关于帮助上下文的内容）。
- IContributedContentsView——被PropertySheet用于标识工作台视图。该视图允许其他部分（一般是活动部分）提供他们的内容。
- IShowInSource——使视图可用。当用户选择Navigate > Show In > ...时提供上下文。
- IShowInTarget——使视图出现于Navigate > Show In子菜单中，并当用户在该子菜单中做出选择时，调用IShowInTarget#show(ShowInContext)方法。
- IShowInTargetList——为应出现于Navigate > Show In子菜单的视图指定标识符。

7.2.2 视图控件

视图可以包含任意类型和数量的控件，但一般情况下，诸如Favorites视图的视图包含一个表和树控件。收藏夹视图可以直接使用SWT表部件（org.eclipse.swt.widgets.Table，参见4.2.6节），然而，更高级别的JFace表查看器（org.eclipse.jface.viewers.TableViewer，参见5.1.7节）包装了SWT表部件并且更容易使用。它处理了大量的繁重的底层工作，允许你直接添加、选择和移除模型对象而不是处理TableItem的底层实例。

了解了这一点之后，让我们从添加三个新字段至FavoritesView类开始：

```
private TableColumn typeColumn;  
private TableColumn nameColumn;  
private TableColumn locationColumn;
```

你应继续改进createPartControl()方法。该方法作为创建Favorites插件（参见2.3.3节）的一部分生成，以使表具有三列。SWT.FULL_SELECTION样式块使得当用户做出选择时，高亮显示整行。

```
viewer = new TableViewer(parent,
    SWT.H_SCROLL | SWT.V_SCROLL | SWT.MULTI | SWT.FULL_SELECTION);
final Table table = viewer.getTable();

typeColumn = new TableColumn(table, SWT.LEFT);
typeColumn.setText("");
typeColumn.setWidth(18);

nameColumn = new TableColumn(table, SWT.LEFT);
nameColumn.setText("Name");
nameColumn.setWidth(200);

locationColumn = new TableColumn(table, SWT.LEFT);
locationColumn.setText("Location");
locationColumn.setWidth(450);

table.setHeaderVisible(true);
table.setLinesVisible(false);

viewer.setContentProvider(new ViewContentProvider());
viewer.setLabelProvider(new ViewLabelProvider());
viewer.setInput(getViewSite());
```

然后，当你想要更进一步改进时，可以让表中的列自动调整大小（参见7.8节）。

7.2.3 视图模型

视图可以具有它自己的内容模型，比如Favorites视图。它可以使用已有的模型对象，比如IResource和它的子类型。它也可以完全不具有一个模型。在这种情况下，创建：

- IFavoriteItem——用于抽象不同类型的Favorites对象的区别的接口。
- FavoritesManager——保存Favorites模型对象。
- FavoriteResource——将资源适配于IFavoriteItem接口的类。
- FavoriteJavaElement——将Java元素适配于IFavoriteItem接口的类。

IFavoriteItem接口隐藏了不同类型的Favorites对象之间的区别。这项功能使得FavoritesManager和FavoritesView可以以一种统一的方式处理所有的Favorites项。下面的用于Eclipse的多个地方的命名约定，是在接口名称前加“I”。因此，该接口的名称是IFavoriteItem而不是所预想的FavoriteItem（参见7.4.2节以了解更多关于IAdaptable的信息）。

```
package com.qualityeclipse.favorites.model;

public interface IFavoriteItem
    extends IAdaptable
{
    String getName();
    void setName(String newName);
    String getLocation();
    boolean isFavoriteFor(Object obj);
    FavoriteItemType getType();
    String getInfo();

    static IFavoriteItem[] NONE = new IFavoriteItem[] {};
}
```

稍后，Favorites项将被序列化，以使它们可以放置于剪贴板（参见7.3.7节），并保存至Eclipse

工作台会话间的磁盘（参见7.5.2节）。最后，每个项的`getInfo()`方法必须返回足够的状态，以便可以在以后正确地重构项。

由`getType()`方法返回的`FavoriteItemType`对象是一个类型安全（type-safe）的枚举。它可以用于排序和存储Favorites对象。它具有一个与其相关的用于显示目的的可读名称。引入`FavoriteItemType`而不是简单的String或int类型允许排序顺序可以被与收藏夹对象类型关联的可读名称所分隔。

```
package com.qualityeclipse.favorites.model;

import ...

public abstract class FavoriteItemType
    implements Comparable
{
    private static final ISharedImages PLATFORM_IMAGES =
        PlatformUI.getWorkbench().getSharedImages();
```

然后，你需要添加一个包含几个字段和访问方法的构造函数至Favorites视图所使用的`FavoriteItemType`以用于排序并显示Favorites项。由于工作台已经为不同类型的资源提供了图像，`FavoriteItemType`对象只简单地返回合适的共享的图像。要为收藏夹对象的其他类型返回自定义图像，你可以在插件的生命周期中缓存这些图像并当插件被关闭时释放它们（参见7.7节）。

```
private final String id;
private final String printName;
private final int ordinal;

private FavoriteItemType(String id, String name, int position) {
    this.id = id;
    this.ordinal = position;
    this.printName = name;
}

public String getId() {
    return id;
}

public String getName() {
    return printName;
}

public abstract Image getImage();
public abstract IFavoriteItem newFavorite(Object obj);
public abstract IFavoriteItem loadFavorite(String info);
```

基于排序的目的，`FavoriteItemType`实现了`Comparable<FavoriteItemType>`接口。因此它还必须实现`compareTo`方法。

```
public int compareTo(FavoriteItemType other) {
    return this.ordinal - other.ordinal;
}
```

然后，为收藏夹的每一个已知类型添加`public static`字段。到目前为止，这些实例是硬编码的，然而，在以后，这些实例将被扩展点定义，以使他人可以引入收藏夹的新类型（参见17.3节）。这些新的公共静态字段基于`org.eclipse.core.resources`、`org.eclipse.ui.ide`和`org.eclipse.jdt.core`插件。所以，可以使用插件清单编辑器的Dependencies页面（图2-10）来添加这些必需的插件，然后保存更改。

```
public static final FavoriteItemType UNKNOWN
    = new FavoriteItemType("Unknown", "Unknown", 0)
{
    public Image getImage() {
        return null;
    }
    public IFavoriteItem newFavorite(Object obj) {
        return null;
    }
    public IFavoriteItem loadFavorite(String info) {
        return null;
    }
};

public static final FavoriteItemType WORKBENCH_FILE
    = new FavoriteItemType("WBFile", "Workbench File", 1)
{
    public Image getImage() {
        return PLATFORM_IMAGES
            .getImage(org.eclipse.ui.ISharedImages.IMG_OBJ_FILE);
    }
    public IFavoriteItem newFavorite(Object obj) {
        if (!(obj instanceof IFile))
            return null;
        return new FavoriteResource(this, (IFile) obj);
    }
    public IFavoriteItem loadFavorite(String info) {
        return FavoriteResource.loadFavorite(this, info);
    }
};

public static final FavoriteItemType WORKBENCH_FOLDER
    = new FavoriteItemType("WBFolder", "Workbench Folder", 2)
{
    public Image getImage() {
        return PLATFORM_IMAGES
            .getImage(org.eclipse.ui.ISharedImages.IMG_OBJ_FOLDER);
    }
    public IFavoriteItem newFavorite(Object obj) {
        if (!(obj instanceof IFolder))
            return null;
        return new FavoriteResource(this, (IFolder) obj);
    }
    public IFavoriteItem loadFavorite(String info) {
        return FavoriteResource.loadFavorite(this, info);
    }
};
... more of the same ...
```

最后，创建一个包含所有已知类型的静态数组和一个将返回所有已知类型的getTypes()方法。

```
private static final FavoriteItemType[] TYPES = {
    UNKNOWN, WORKBENCH_FILE, WORKBENCH_FOLDER, WORKBENCH_PROJECT,
    JAVA_PROJECT, JAVA_PACKAGE_ROOT, JAVA_PACKAGE,
    JAVA_CLASS_FILE, JAVA_COMP_UNIT, JAVA_INTERFACE, JAVA_CLASS};

public static FavoriteItemType[] getTypes() {
```

```
    return TYPES;
}
```

所有Favorites视图应显示同样的Favorites对象集，因此，FavoritesManager是一个用于维持该全局集合的单一元素集。

```
package com.qualityeclipse.favorites.model;

import ...

public class FavoritesManager {
    private static FavoritesManager manager;
    private Collection<IFavoriteItem> favorites;

    private FavoritesManager() {}

    public static FavoritesManager getManager() {
        if (manager == null)
            manager = new FavoritesManager();
        return manager;
    }

    public IFavoriteItem[] getFavorites() {
        if (favorites == null)
            loadFavorites();
        return favorites.toArray(new IFavoriteItem[favorites.size()]);
    }

    private void loadFavorites() {
        // temporary implementation
        // to prepopulate list with projects
        IProject[] projects = ResourcesPlugin.getWorkspace().getRoot()
            .getProjects();
        favorites = new HashSet(projects.length);
        for (int i = 0; i < projects.length; i++)
            favorites.add(new FavoriteResource(
                FavoriteItemType.WORKBENCH_PROJECT, projects[i]));
    }
}
```

管理器需要在已有的Favorites对象中查找并创建新的Favorites对象。

```
public void addFavorites(Object[] objects) {
    if (objects == null)
        return;
    if (favorites == null)
        loadFavorites();
    Collection<IFavoriteItem> items =
        new HashSet<IFavoriteItem>(objects.length);
    for (int i = 0; i < objects.length; i++) {
        IFavoriteItem item = existingFavoriteFor(objects[i]);
        if (item == null) {
            item = newFavoriteFor(objects[i]);
            if (item != null && favorites.add(item))
                items.add(item);
        }
    }
    if (items.size() > 0) {
        IFavoriteItem[] added =
```

```
        items.toArray(new IFavoriteItem[items.size()]);
        fireFavoritesChanged(added, IFavoriteItem.NONE);
    }
}

public void removeFavorites(Object[] objects) {
    if (objects == null)
        return;
    if (favorites == null)
        loadFavorites();
    Collection<IFavoriteItem> items =
        new HashSet<IFavoriteItem>(objects.length);
    for (int i = 0; i < objects.length; i++) {
        IFavoriteItem item = existingFavoriteFor(objects[i]);
        if (item != null && favorites.remove(item))
            items.add(item);
    }
    if (items.size() > 0) {
        IFavoriteItem[] removed =
            items.toArray(new IFavoriteItem[items.size()]);
        fireFavoritesChanged(IFavoriteItem.NONE, removed);
    }
}

public IFavoriteItem newFavoriteFor(Object obj) {
    FavoriteItemType[] types = FavoriteItemType.getTypes();
    for (int i = 0; i < types.length; i++) {
        IFavoriteItem item = types[i].newFavorite(obj);
        if (item != null)
            return item;
    }
    return null;
}

private IFavoriteItem existingFavoriteFor(Object obj) {
    if (obj == null)
        return null;
    if (obj instanceof IFavoriteItem)
        return (IFavoriteItem) obj;
    Iterator<IFavoriteItem> iter = favorites.iterator();
    while (iter.hasNext()) {
        IFavoriteItem item = iter.next();
        if (item.isFavoriteFor(obj))
            return item;
    }
    return null;
}

public IFavoriteItem[] existingFavoritesFor(Iterator<?> iter) {
    List<IFavoriteItem> result = new ArrayList<IFavoriteItem>(10);
    while (iter.hasNext()) {
        IFavoriteItem item = existingFavoriteFor(iter.next());
        if (item != null)
            result.add(item);
    }
    return (IFavoriteItem[]) result.toArray(
```

```

        new IFavoriteItem[result.size()]);
    }

```

由于多个视图将访问这些信息，管理程序必须可以在信息改变时通知注册的监听器。FavoritesManager将只能从UI线程访问，所以你不需担心线程安全（参见4.2.5以了解更多关于UI线程）。

```

private List<FavoritesManagerListener> listeners
    = new ArrayList<FavoritesManagerListener>();

public void addFavoritesManagerListener(
    FavoritesManagerListener listener
) {
    if (!listeners.contains(listener))
        listeners.add(listener);
}

public void removeFavoritesManagerListener(
    FavoritesManagerListener listener
) {
    listeners.remove(listener);
}

private void fireFavoritesChanged(
    IFavoriteItem[] itemsAdded, IFavoriteItem[] itemsRemoved
) {
    FavoritesManagerEvent event = new FavoritesManagerEvent(
        this, itemsAdded, itemsRemoved);
    for (Iterator<FavoritesManagerListener>
        iter = listeners.iterator(); iter.hasNext();)
        iter.next().favoritesChanged(event);
}

```

FavoritesManager使用FavoritesManagerListener和FavoritesManagerEvent类以将更改通知感兴趣的对象。

```

package com.qualityeclipse.favorites.model;

public interface FavoritesManagerListener
{
    public void favoritesChanged(FavoritesManagerEvent event);
}

package com.qualityeclipse.favorites.model;

import java.util.EventObject;

public class FavoritesManagerEvent extends EventObject
{
    private static final long serialVersionUID = 3697053173951102953L;

    private final IFavoriteItem[] added;
    private final IFavoriteItem[] removed;

    public FavoritesManagerEvent(
        FavoritesManager source,
        IFavoriteItem[] itemsAdded, IFavoriteItem[] itemsRemoved
    ) {
        super(source);
    }
}

```

```
    ) {  
        super(source);  
        added = itemsAdded;  
        removed = itemsRemoved;  
    }  
  
    public IFavoriteItem[] getItemsAdded() {  
        return added;  
    }  
  
    public IFavoriteItem[] getItemsRemoved() {  
        return removed;  
    }  
}
```

在以后, FavoritesManager将被改进以允许该列表在Eclipse会话间可以保持持久性(参见7.5.2节)。但就现在而言, 该列表将在每一次Eclipse启动时使用当前工作区项目进行初始化。此外, 当前实现将在后续几章得到扩展以包含由其他插件添加的Favorites类型(参见17.3节)。

FavoriteResource包装了一个IResource对象, 将其适配于IFavoriteItem接口。为了了解更多关于下面引用的IAdaptable接口的内容, 参见21.3节。

```
package com.qualityeclipse.favorites.model;  
  
import ...  
  
public class FavoriteResource  
    implements IFavoriteItem  
{  
    private FavoriteItemType type;  
    private IResource resource;  
    private String name;  
    FavoriteResource(FavoriteItemType type, IResource resource) {  
        this.type = type;  
        this.resource = resource;  
    }  
  
    public static FavoriteResource loadFavorite(  
        FavoriteItemType type, String info)  
    {  
        IResource res = ResourcesPlugin.getWorkspace().getRoot()  
            .findMember(new Path(info));  
        if (res == null)  
            return null;  
        return new FavoriteResource(type, res);  
    }  
  
    public String getName() {  
        if (name == null)  
            name = resource.getName();  
        return name;  
    }  
  
    public void setName(String newName) {  
        name = newName;  
    }  
  
    public String getLocation() {  
        IPath path = resource.getLocation().removeLastSegments(1);  
        if (path.segmentCount() == 0)  
            return "";  
    }  
}
```



```

        return path.toString();
    }

    public boolean isFavoriteFor(Object obj) {
        return resource.equals(obj);
    }

    public FavoriteItemType getType() {
        return type;
    }

    public boolean equals(Object obj) {
        return this == obj || (
            (obj instanceof FavoriteResource)
            && resource.equals(((FavoriteResource) obj).resource));
    }

    public int hashCode() {
        return resource.hashCode();
    }

    public Object getAdapter(Class adapter) {
        if (adapter.isInstance(resource))
            return resource;
        return Platform.getAdapterManager().getAdapter(this, adapter);
    }

    public String getInfo() {
        return resource.getFullPath().toString();
    }
}

```

与FavoriteResource类似, FavoriteJavaElement将一个IJavaElement对象适配于IFavoriteItem接口。在创建该类之前, 你将需要添加org.eclipse.jdt.ui插件至Favorites插件清单(图7-5)。

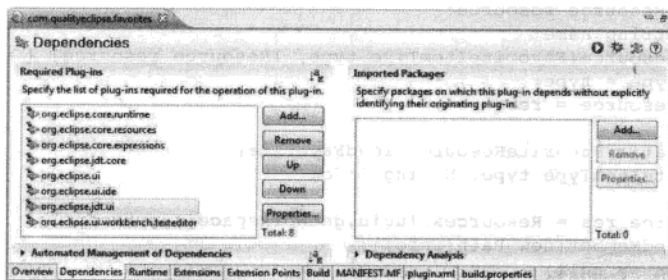


图7-5 插件清单编辑器的Dependencies页面

如果项目是插件项目(参见2.2节), 修改插件清单将使得项目的Java创建路径得到自动更新。

```
package com.qualityeclipse.favorites.model;
```

```
import ...

public class FavoriteJavaElement
    implements IFavoriteItem
{
    private FavoriteItemType type;
    private IJavaElement element;
    private String name;

```

```
public FavoriteJavaElement(
    FavoriteItemType type, IJavaElement element
) {
    this.type = type;
    this.element = element;
}
public static FavoriteJavaElement loadFavorite(
    FavoriteItemType type, String info
) {
    IResource res = ResourcesPlugin.getWorkspace().getRoot()
        .findMember(new Path(info));
    if (res == null)
        return null;
    IJavaElement elem = JavaCore.create(res);
    if (elem == null)
        return null;
    return new FavoriteJavaElement(type, elem);
}
public String getName() {
    if (name == null)
        name = element.getElementName();
    return name;
}
public void setName(String newName) {
    name = newName;
}
public String getLocation() {
    try {
        IResource res = element.getUnderlyingResource();
        if (res != null) {
            IPath path = res.getLocation().removeLastSegments(1);
            if (path.segmentCount() == 0)
                return "";
            return path.toString();
        }
    }
    catch (JavaModelException e) {
        FavoritesLog.logError(e);
    }
    return "";
}
public boolean isFavoriteFor(Object obj) {
    return element.equals(obj);
}
public FavoriteItemType getType() {
    return type;
}
public boolean equals(Object obj) {
    return this == obj || (
        (obj instanceof FavoriteJavaElement)
        && element.equals(((FavoriteJavaElement) obj).element));
}
public int hashCode() {
    return element.hashCode();
}
public Object getAdapter(Class adapter) {
```

```

        if (adapter.isInstance(element))
            return element;
        IResource resource = element.getResource();
        if (adapter.isInstance(resource))
            return resource;
        return Platform.getAdapterManager().getAdapter(this, adapter);
    }

    public String getInfo() {
        try {
            return element.getUnderlyingResource().getFullPath()
                .toString();
        }
        catch (JavaModelException e) {
            FavoritesLog.logError(e);
            return null;
        }
    }
}

```

7.2.4 内容提供者

当创建了模型对象后，它们需要被链接至视图。内容提供者负责从输入对象释放对象（在这里是 FavoritesManager），并将它们传递至表查看器以一行一个对象样式显示。虽然 IStructuredContentProvider 没有指定，内容提供者也还负责当 FavoritesManager 更改时更新查看器。

在获取作为 FavoritesView 类的一部分自动生成的内容提供者（参见 2.3.2 节）和修改它使用新创建的 FavoritesManager 之后，它将和以下代码类似：

```

package com.qualityeclipse.favorites.views;

import ...

class FavoritesViewContentProvider
    implements IStructuredContentProvider, FavoritesManagerListener
{
    private TableViewer viewer;
    private FavoritesManager manager;
    public void inputChanged(
        Viewer viewer, Object oldInput, Object newInput
    ) {
        this.viewer = (TableViewer) viewer;
        if (manager != null)
            manager.removeFavoritesManagerListener(this);
        manager = (FavoritesManager) newInput;
        if (manager != null)
            manager.addFavoritesManagerListener(this);
    }
    public void dispose() {
    }

    public Object[] getElements(Object parent) {
        return manager.getFavorites();
    }
    public void favoritesChanged(FavoritesManagerEvent event) {
        viewer.getTable().setRedraw(false);
    }
}

```

```
try {
    viewer.remove(event.getItemsRemoved());
    viewer.add(event.getItemsAdded());
}
finally {
    viewer.getTable().setRedraw(true);
}
}
```

提示 上面的方法使用了setRedraw方法以当从查看器中添加和移除多个项目时降低闪烁。

获取并修改内容提供者表示在createPartControl方法中的对于setContentProvider和setInput已经更改为以下内容：

```
viewer.setContentProvider(new FavoritesViewContentProvider());
viewer.setInput(FavoritesManager.getManager());
```

7.2.5 标签提供者

标签提供者占据由内容提供者返回的一个表的行对象，并获取将在列中显示的值。在重构FavoritesView.ViewLabelProvider内部类（参见2.3.3节）为一个最高级类并修改它以获取来源于新创建的模型对象的值时，它和以下代码类似。

```
class FavoritesViewLabelProvider extends LabelProvider
    implements ITableLabelProvider
{
    public String getColumnText(Object obj, int index) {
        switch (index) {
            case 0: // Type column
                return "";
            case 1: // Name column
                if (obj instanceof IFavoriteItem)
                    return ((IFavoriteItem) obj).getName();
                if (obj != null)
                    return obj.toString();
                return "";
            case 2: // Location column
                if (obj instanceof IFavoriteItem)
                    return ((IFavoriteItem) obj).getLocation();
                return "";
            default:
                return "";
        }
    }
    public Image getColumnImage(Object obj, int index) {
        if ((index == 0) && (obj instanceof IFavoriteItem))
            return ((IFavoriteItem) obj).getType().getImage();
        return null;
    }
}
```

要使用不同的字体和颜色改进Favorites视图，请分别实现IFontProvider和IColorProvider（参见13.2.5节）。

提示 如果你正显示工作台相关对象，WorkbenchLabelProvider和WorkbenchPartLabelProvider包含用于指定实现IWorkbenchAdapter接口的工作台资源的相关文本和图像的行为（参见21.3.4节）。对于列表和单列树、单列表，可以实现IViewerLabelProvider以通过实现单个updateLabel()方法有效的设置文本、图像、字体和颜色。

7.2.6 查看器排序器

虽然内容提供者提供行对象，在行对象被显示之前，对这些行对象进行排序属于ViewerSorter的责任。在Favorites视图中，当前有三种条件以用于升序或降序排序项目：

- 名称 (Name)
- 类型 (Type)
- 位置 (Location)

FavoritesViewSorter代表根据三种排序策略，每种对应于上面列出的条件。此外，FavoritesViewSorter在列标题中监听鼠标点击并根据被点击的列对表内容进行重新排列。再次点击一列将切换排列顺序。

```
package com.qualityeclipse.favorites.views;

import ...

public class FavoritesViewSorter extends ViewerSorter
{
    // Simple data structure for grouping
    // sort information by column.
    private class SortInfo {
        int columnIndex;
        Comparator<Object> comparator;
        boolean descending;
    }
    private TableViewer viewer;
    private SortInfo[] infos;

    public FavoritesViewSorter(
        TableViewer viewer,
        TableColumn[] columns,
        Comparator<Object>[] comparators
    ) {
        this.viewer = viewer;
        infos = new SortInfo[columns.length];
        for (int i = 0; i < columns.length; i++) {
            infos[i] = new SortInfo();
            infos[i].columnIndex = i;
            infos[i].comparator = comparators[i];
            infos[i].descending = false;
            createSelectionListener(columns[i], infos[i]);
        }
    }
    public int compare(
        Viewer viewer, Object favorite1, Object favorite2
    ) {
        for (int i = 0; i < infos.length; i++) {
```

```

        int result = infos[i].comparator
            .compare(favorite1, favorite2);
        if (result != 0) {
            if (infos[i].descending)
                return -result;
            return result;
        }
    }
    return 0;
}
private void createSelectionListener(
    final TableColumn column, final SortInfo info
) {
    column.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            sortUsing(info);
        }
    });
}
protected void sortUsing(SortInfo info) {
    if (info == infos[0])
        info.descending = !info.descending;
    else {
        for (int i = 0; i < infos.length; i++) {
            if (info == infos[i]) {
                System.arraycopy(infos, 0, infos, 1, i);
                infos[0] = info;
                info.descending = false;
                break;
            }
        }
    }
    viewer.refresh();
}
}

```

现在, FavoritesView将引入一个新的字段以保存排序器实例:

```
private FavoritesViewSorter sorter;
```

而Favorites视图createPartControl(Composite)方法将被修改以调用如下所示的新的方法。然后, 当前被用户所选择的排序顺序, 必须在Eclipse会话间保存 (参见7.5.1节)。

```

private void createTableSorter() {
    Comparator<IFavoriteItem> nameComparator
        = new Comparator<IFavoriteItem>() {
        public int compare(IFavoriteItem i1, IFavoriteItem i2) {
            return i1.getName().compareTo(i2.getName());
        }
    };
    Comparator<IFavoriteItem> locationComparator
        = new Comparator<IFavoriteItem>() {
        public int compare(IFavoriteItem i1, IFavoriteItem i2) {
            return i1.getLocation().compareTo(i2.getLocation());
        }
    };
    Comparator<IFavoriteItem> typeComparator
        = new Comparator<IFavoriteItem>() {

```

```

        public int compare(IFavoriteItem i1, IFavoriteItem i2) {
            return i1.getType().compareTo(i2.getType());
        }
    };
    sorter = new FavoritesViewSorter(
        viewer,
        new TableColumn[] {
            nameColumn, locationColumn, typeColumn },
        new Comparator[] {
            nameComparator, locationComparator, typeComparator }
    );
    viewer.setSorter(sorter);
}

```

7.2.7 查看器过滤器

ViewerFilter的子类决定由内容提供者返回的哪些行对象将被显示，而哪些不会被显示。由于将有可能只有一个内容提供者、一个标签提供者和一个排序器，然而可以有任意数量的过滤器与视图相关联。当有多个过滤器适用，只有那些满足所有适用过滤器的项才会被显示。

与刚讨论过的排序类似，Favorites视图可以基于以下条件过滤：

- 名称 (Name)
- 类型 (Type)
- 位置 (Location)

Eclipse提供了org.eclipse.ui.internal.misc.StringMatcher类型，对于通配符过滤是理想的选择，但是由于该类是一个内部包，第一步是将该类复制至com.qualityeclipse.favorites.util包。虽然复制听起来是可怕的，但在Eclipse内部的不同地方，已经有了该类的10个副本。所有都是内部的（参见21.2节以了解更多关于内部包和相关话题的内容）。

当完成了这项内容后，用于使用名称过滤Favorites视图ViewerFilter类将与此类似（见下面的代码）。该查看器过滤器通过使用7.3.4节中的命令与Favorites视图相关联。

```

package com.qualityeclipse.favorites.views;

import ...

public class FavoritesViewNameFilter extends ViewerFilter
{
    private final StructuredViewer viewer;
    private String pattern = "";
    private StringMatcher matcher;
    public FavoritesViewNameFilter(StructuredViewer viewer) {
        this.viewer = viewer;
    }
    public String getPattern() {
        return pattern;
    }
    public void setPattern(String newPattern) {
        boolean filtering = matcher != null;
        if (newPattern != null && newPattern.trim().length() > 0) {
            pattern = newPattern;
            matcher = new StringMatcher(pattern, true, false);
            if (!filtering)
                viewer.addFilter(this);
        }
    }
}

```

```
        else
            viewer.refresh();
    }
    else {
        pattern = "";
        matcher = null;
        if (filtering)
            viewer.removeFilter(this);
    }
}

public boolean select(
    Viewer viewer,
    Object parentElement,
    Object element
) {
    return matcher.match(
        ((IFavoriteItem) element).getName());
}
}
```

7.2.8 视图选择

一旦完成了模型对象和视图控件，视图的其他方面，特别是命令和操作，需要一种确定当前哪些Favorites项被选中的方法。将以下方法添加至FavoritesView以使操作可以在选中项上执行。

```
public IStructuredSelection getSelection() {
    return (IStructuredSelection) viewer.getSelection();
}
```

7.2.9 实现propertyTester

当模型存在时，我们就可以完成实现首先6.2.5节中的propertyTester。修改FavoritesTester以测试FavoritesManager是否包含指定的对象：

```
public boolean test(Object receiver, String property, Object[] args,
    Object expectedValue) {

    boolean found = false;
    IFavoriteItem[] favorites
        = FavoritesManager.getManager().getFavorites();
    for (int i = 0; i < favorites.length; i++) {
        IFavoriteItem item = favorites[i];
        found = item.isFavoriteFor(receiver);
        if (found)
            break;
    }
    if ("isFavorite".equals(property))
        return found;
    if ("notFavorite".equals(property))
        return !found;
    return false;
}
```

7.3 视图命令

视图命令可以有以下几种显示形式：视图上下文菜单的菜单项、视图标题栏右侧的工具栏按钮

和视图下拉菜单的菜单项（图6-16）。本节包含了在程序中添加命令至视图和注册该视图以使他人可以通过插件清单添加他们自己的命令和操作。相比之下，6.1节和6.8节讨论了使用插件清单中的声明来添加命令和操作。

7.3.1 模型命令处理器

既然已经完成了模型对象，我们就可以完成在6.3.1节中介绍的AddToFavoritesHandler类（实现6.7.3节中介绍的AddToFavoritesActionDelegate与此十分类似）了。使用下面列出的更改，处理器添加选中项至FavoritesManager。然后，FavoritesManager通知FavoritesViewContentProvider，并由FavoritesViewContentProvider刷新表以显示新的信息。

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection)
        FavoritesManager.getManager().addFavorites(
            ((IStructuredSelection) selection).toArray());
    return null;
}
```

7.3.2 上下文菜单

一般地，视图具有由关注视图或其中的选中项的命令生成的上下文菜单。在程序中创建视图上下文菜单包含几个步骤。如果你想要其他插件通过插件清单中的声明（参见6.2.5节）添加命令至你的视图上下文，那么你必须使用更多的几个步骤以注册你的视图。

1. 创建添加项

第一步是创建将出现于上下文菜单的添加项。对于Favorites视图来说，将选中元素从视图中移除是必要的。如果isDynamic()返回true，那么当每一次显示上下文菜单时，而不是上下文菜单第一次显示时，都将会调用fill(...)。这当你的操作更改可见性时是有用的。在我们的示例中，我们不覆盖isDynamic()，这是因为我们的操作不改变可见性而改变可用性。

```
package com.qualityeclipse.favorites.contributions;

import ...

public class RemoveFavoritesContributionItem
    extends ContributionItem
{
    private final FavoritesView view;
    private final IHandler handler;
    boolean enabled = false;
    private MenuItem menuItem;

    public RemoveFavoritesContributionItem(FavoritesView view,
        IHandler handler) {
        this.view = view;
        this.handler = handler;
        view.addSelectionChangedListener(
            new ISelectionChangedListener() {
                public void selectionChanged(SelectionChangedEvent event) {
                    enabled = !event.getSelection().isEmpty();
                    updateEnablement();
                }
            }
        );
    }
}
```

```

    }
    });
}
public void fill(Menu menu, int index) {
    menuItem = new MenuItem(menu, SWT.NONE, index);
    menuItem.setText("Remove");
    menuItem.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            run();
        }
    });
    updateEnablement();
}
private void updateEnablement() {
    Image image =
        PlatformUI.getWorkbench().getSharedImages().getImage(
            enabled ? ISharedImages.IMG_TOOL_DELETE
                : ISharedImages.IMG_TOOL_DELETE_DISABLED);
    if (menuItem != null) {
        menuItem.setImage(image);
        menuItem.setEnabled(enabled);
    }
}
public void run() {
    final IHandlerService handlerService = (IHandlerService)
        viewSite.getService(IHandlerService.class);
    IEvaluationContext evaluationContext =
        handlerService.createContextSnapshot(true);
    ExecutionEvent event =
        new ExecutionEvent(null, Collections.EMPTY_MAP, null,
            evaluationContext);
    try {
        handler.execute(event);
    }
    catch (ExecutionException e) {
        FavoritesLog.logError(e);
    }
}
}
}

```

提示 通过插件清单为6.2.6节中的每一个项定义提供一个命令比起上面所示的创建添加项的方法更为容易。除非你需要比创建你的添加项所能提供的灵活性更高的灵活性。

RemoveFavoritesContributionItem使用了新的处理器以执行操作。我们将这项功能从添加项分离出来以使它能在其他地方被优化。

```

public class RemoveFavoritesHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        ISelection selection = HandlerUtil.getCurrentSelection(event);
        if (selection instanceof IStructuredSelection)
            FavoritesManager.getManager().removeFavorites(
                ((IStructuredSelection) selection).toArray());
        return null;
    }
}

```

在FavoritesView类中，为添加项提供访问方法以更新它的可用状态：

```
public void addSelectionChangedListener(
    ISelectionChangedListener listener) {
    viewer.addSelectionChangedListener(listener);
}
```

并创建新的字段：

```
private IHandler removeHandler;
private RemoveFavoritesContributionItem removeContributionItem;
```

然后从createPartControl(Composite)调用以下新的方法以初始化该字段。

```
private void createContributions() {
    removeHandler = new RemoveFavoritesHandler();
    removeContributionItem =
        new RemoveFavoritesContributionItem(getViewSite(),
            removeHandler);
}
```

选中的菜单项 为了了解具有选中标记的动态被添加的菜单项，参见14.3.7节。

2. 创建上下文菜单

上下文菜单必须与视图同时创建，但是由于添加者基于当前选择添加或移除菜单项，它的内容将直到当用户点击了鼠标右键之后，菜单显示之前，才能被确定。为了完成这项任务，将菜单的RemoveAllWhenShown属性设置为true，以使菜单每次都将从头开始创建，然后添加菜单监听器以动态创建菜单。此外，菜单必须注册至控件，以使它能被显示和与视图站点一起使用，这样其他插件可以向它添加项操作（参见6.2.5节）。对于Favorites视图来说，修改createPartControl()以调用以下新的createContextMenu()方法。

```
private void createContextMenu() {
    MenuManager menuMgr = new MenuManager("#PopupMenu");
    menuMgr.setRemoveAllWhenShown(true);
    menuMgr.addMenuListener(new IMenuListener() {
        public void menuAboutToShow(IMenuManager m) {
            FavoritesView.this.fillContextMenu(m);
        }
    });
    Menu menu =
        menuMgr.createContextMenu(viewer.getControl());
    viewer.getControl().setMenu(menu);
    getSite().registerContextMenu(menuMgr, viewer);
}
```

3. 动态创建上下文菜单

每当用户点击鼠标右键时，必须从头开始重建上下文菜单的内容。这是因为添加者可以基于选中项添加或移除菜单项。此外，上下文菜单必须包含一个具有IWorkbench ActionConstants.MB_ADDITIONS常量的分割线，用于表示被添加的操作可以出现于菜单的位置。createContextMenu()方法（参见7.3.2节）调用了这里所示的新的fillContextMenu(IMenuManager)方法：

```
private void fillContextMenu(IMenuManager menuMgr) {
    menuMgr.add(removeContributionItem);
    menuMgr.add(new Separator(
        IWorkbenchActionConstants.MB_ADDITIONS));
}
```

提示 在fillContextMenu中添加Separators和GroupMarkers（参见以上内容7.3.7节）以更准确地指定通过插件清单添加项的菜单命令将出现的位置（参见6.2.6节）。

4. 选择提供者

当定义了基于选择的添加项（参见6.2.5节）后，它们将用于针对选中项而不是视图。要使基于选择的添加项出现于视图的上下文菜单中，不仅视图要注册上下文菜单（参见7.3.2节），它还必须将它的选择向所有其他已注册的监听器发布（参见7.4.1节）。此外，基于选择的关系一般针对特定类型的对象而不是所有对象。这意味着添加者可以将选中对象适配于任意可以查询并操作的对象（参见7.4.2节）。

5. 过滤不需要的操作

如果视图注册了它的上下文菜单（参见7.3.2节），那么所有具有locationURI “popup.org.eclipse.ui.popup.any”的菜单添加项将出现在该视图的上下文菜单中。此时，由于已经定义了Favorites使用visibleWhen表达式的方式，Favorites子菜单将出现于所有它应出现的视图中，除了Favorites视图之外。如果不是这种情况，压缩Favorites子菜单使其不显示于Favorites视图中的一种方法将是修改该菜单添加项的visibleWhen表达式：

```
<visibleWhen checkEnabled="false">
  <and>
    <not>
      <with variable="activePartId">
        <equals value =
          "com.qualityeclipse.favorites.views.FavoritesView">
        </equals>
      </with>
    </not>
    <with variable="selection">
      <iterate ...
```

插件编辑器将不会允许你插入<and>XML元素并作为根元素于Extensions页中一个已有的visibleWhen页面。作为替代，你必须切换至plugin.xml页面并编辑该XML表达式本身。

7.3.3 工具栏按钮

接下来，在程序中添加Remove操作至工具栏（参见6.8.4以了解关于使用插件清单而不是在程序中声明工具栏按钮）。此外，该工具栏按钮的状态需要基于Favorites视图中的选择而改变。在FavoritesView类中，从createPartControl(Composite)方法调用以下新方法：

```
private void createToolBarButtons() {
    IToolBarManager toolBarMgr =
        getViewSite().getActionBars().getToolBarManager();
    toolBarMgr.add(new GroupMarker("edit"));
    toolBarMgr.add(removeContributionItem);
}
```

在RemoveFavoritesContributionItem中，添加一个附加字段

```
private ToolItem toolItem;
```

并添加以下新方法。

```
public void fill(ToolBar parent, int index) {
```

```

toolItem = new ToolItem(parent, SWT.NONE, index);
toolItem.setToolTipText("Remove the selected favorite items");
toolItem.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        run();
    }
});
updateEnablement();
}

```

工具栏项必须基于视图的当前选择来设置其可用状态，因此将以下代码添加至updateEnablement(...)方法。

```

if (toolItem != null) {
    toolItem.setImage(image);
    toolItem.setEnabled(enabled);
}

```

提示 作为替代，对于你自己的视图的工具栏的添加项使用了插件清单中的声明。为了了解更多关于menuContribution和locationURI的内容，参见6.2.6节和6.2.9节。

7.3.4 下拉菜单

本节将在程序中添加操作至Favorites视图下拉菜单，使得名称过滤器可以被设置其可用状态（参见6.8.5节以了解关于在插件清单中而不是在程序中定义下拉菜单项）。我们可以和上一节中创建RemoveFavoritesContributionItem的子类一样，创建一个ContributionItem的子类，但是，作为替代，我们继承Action以说明较老的方法。现在，操作将会使用一个简单的InputDialog以提示输入名称过滤器样式，但是，这在本书后续内容中将被一个特定的Favorites视图过滤器对话框所取代（参见11.1.2节）。

```

package com.qualityeclipse.favorites.views;

import ...

public class FavoritesViewFilterAction extends Action {

    private final Shell shell;

    private final FavoritesViewNameFilter nameFilter;
    public FavoritesViewFilterAction(
        StructuredViewer viewer,
        String text
    ) {
        super(text);
        shell = viewer.getControl().getShell();
        nameFilter = new FavoritesViewNameFilter(viewer);
    }

    public void run() {
        InputDialog dialog = new InputDialog(
            shell,
            "Favorites View Filter",
            "Enter a name filter pattern"
            + " (* = any string, ? = any character)"
            + System.getProperty("line.separator")
            + "or an empty string for no filtering:",

```

```

        nameFilter.getPattern(),
        null);
    if (dialog.open() == InputDialog.OK)
        nameFilter.setPattern(dialog.getValue().trim());
    }
}

```

createPartControl()方法变得很长并在重构中是必需的。在将表列作为字段释放和释放表创建并排序至分开的方法后,将修改createPartControl()方法以调用新的createViewPulldownMenu()方法。该新方法在程序中创建并初始化了filter字段,并添加新的过滤操作至Favorites视图的下拉菜单(图7-6)。

```

private FavoritesViewFilterAction filterAction;

private void createViewPulldownMenu() {
    IMenuManager menu =
        getViewSite().getActionBars().getMenuManager();
    filterAction =
        new FavoritesViewFilterAction(viewer, "Filter...");
    menu.add(filterAction);
}

```

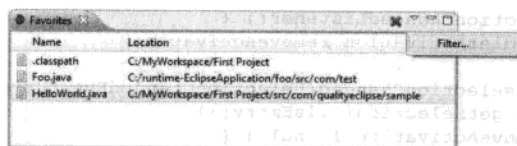


图7-6 显示视图下拉菜单的收藏夹视图

7.3.5 键盘命令

相对于使用鼠标激活上下文菜单然后选择Remove命令从Favorites视图中移除项(参见7.3.2节),仅仅按下Delete键将会更快。这种方法在程序中将Delete键和RemoveFavoritesContributionItem关联起来,而不是如同6.4节中描述的那样通过插件清单定义命令。要使这种办法发挥作用,从createPartControl()方法调用以下hookKeyboard()方法。

```

private void hookKeyboard() {
    viewer.getControl().addKeyListener(new KeyAdapter() {
        public void keyReleased(KeyEvent event) {
            handleKeyReleased(event);
        }
    });
}

protected void handleKeyReleased(KeyEvent event) {
    if (event.character == SWT.DEL && event.stateMask == 0) {
        removeContributionItem.run();
    }
}

```

7.3.6 全局命令

当RemoveFavoritesContributionItem可以通过上下文菜单(参见7.3.2节)和按下Delete键(参见7.3.5节)触发后,同一个添加项需要当用户从Edit菜单中选择Delete也能触发。org.eclipse.ui.

texteditor.IWorkbenchActionDefinitionIds接口为该用途定义了几个常量，比如以下内容。

- 撤销
- 重做
- 剪切
- 复制
- 粘贴
- 删除

当Favorites视图活动时，从createPartControl()方法调用以下新方法将Edit > Delete与RemoveFavoritesAction关联起来。IWorkbenchActionDefinitionIds接口位于org.eclipse.ui.workbench.texteditor插件中，因此它必须在类可以被访问之前添加至插件清单的依赖项。选择监听器激活基于视图当前选择的处理器并使其无效。

```
private void hookGlobalHandlers() {
    IHandlerService handlerService =
        (IHandlerService) getViewSite().getService(
            IHandlerService.class);
    viewer.addSelectionChangedListener(
        new ISelectionChangedListener() {
            private IHandlerActivation removeActivation;

            public void selectionChanged(SelectionChangedEvent event) {
                if (event.getSelection().isEmpty()) {
                    if (removeActivation != null) {
                        handlerService.deactivateHandler(removeActivation);
                        removeActivation = null;
                    }
                }
                else {
                    if (removeActivation == null) {
                        removeActivation =
                            handlerService.activateHandler(
                                IWorkbenchActionDefinitionIds.DELETE,
                                removeHandler);
                    }
                }
            }
        }
    );
}
```

作为替代，你可以使用插件清单中的声明将全局编辑菜单项（参见6.3节）和IWorkbenchActionDefinitionIds常量关联起来。参见7.3.7节。

7.3.7 剪贴板命令

剪贴板相关的三个操作是剪切、复制和粘贴。对于Favorites视图，你需要提供使用三个分开操作剪切选中项目至视图外部，复制选中项和粘贴新项目至视图的功能。

1. 复制

复制操作将选中的Favorites项转换为不同的格式，如资源，并将该信息放置于剪贴板中。我们从创建一个新的CopyFavoritesHandler类开始。该类具有一个方法以用于安全的创建并释放剪贴板对象。剪贴板对象仅存在于对第二个execute(...)方法的调用期间，在此之后，它将被释放。

```
public class CopyFavoritesHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        Clipboard clipboard =
            new Clipboard(HandlerUtil.getActiveShell(event)
                .getDisplay());

        try {
            return execute(event, clipboard);
        }
        finally {
            clipboard.dispose();
        }
    }
}
```

传输的对象将不同格式，如资源，转换为平台特定的字节流和反向转换，以使信息可以在不同的程序间传递（参见7.3.8节以了解更多关于传输类型的内容）。下列CopyFavoritesHandler方法将收藏夹项转换为资源和文本。

```
public static IResource[] asResources(Object[] objects) {
    Collection<IResource> resources =
        new HashSet<IResource>(objects.length);
    for (int i = 0; i < objects.length; i++) {
        Object each = objects[i];
        if (each instanceof IAdaptable) {
            IResource res = (IResource)
                ((IAdaptable) each).getAdapter(IResource.class);
            if (res != null)
                resources.add(res);
        }
    }
    return resources.toArray(new IResource[resources.size()]);
}

public static String asText(Object[] objects) {
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < objects.length; i++) {
        Object each = objects[i];
        if (each instanceof IFavoriteItem) {
            buf.append("Favorite: ");
            buf.append(((IFavoriteItem) each).getName());
        }
        else if (each != null)
            buf.append(each.toString());
        buf.append(System.getProperty("line.separator"));
    }
    return buf.toString();
}
```

最后，CopyFavoritesHandler execute(...)方法执行了实际操作。

```
protected Object execute(ExecutionEvent event, Clipboard clipboard)
    throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection) {
        Object[] objects =
```



```

        ((IStructuredSelection) selection).toArray();
    if (objects.length > 0) {
        try {
            clipboard.setContents(
                new Object[] {
                    asResources(objects), asText(objects), },
                new Transfer[] {
                    ResourceTransfer.getInstance(),
                    TextTransfer.getInstance(), });
        }
        catch (SWTError error) {
            // Copy to clipboard failed.
        }
    }
}
return null;
}

```

该复制处理器将从全局级别关联至Edit > Copy命令，并从本地级别关联至Favorites视图上下文菜单中的新Copy命令。在插件清单中声明复制处理器（参见6.3节）以将其关联至全局的Edit > Copy命令。用于Edit菜单的命令标识符位于org.eclipse.ui.texteditor.IWorkbenchActionDefinitionIds。与6.2.10节中描述的表达式类似，activeWhen表达式指定仅当Favorites视图是活动的，处理器才是活动的。而enabledWhen表达式指定了处理器仅当有一个或多个对象被选中时才可用。

```

<handler
    commandId="org.eclipse.ui.edit.copy"
    class=
        "com.qualityeclipse.favorites.handlers.CopyFavoritesHandler">
    <activeWhen>
        <with variable="activePartId">
            <equals value=
                "com.qualityeclipse.favorites.views.FavoritesView" />
        </with>
    </activeWhen>
    <enabledWhen>
        <with variable="selection">
            <count value="+" />
        </with>
    </enabledWhen>
</handler>

```

2. 上下文菜单中的复制命令

我们还需要Copy命令出现于Favorites视图的上下文菜单中。我们需要剪切、复制和粘贴命令紧挨着出现于上下文中。因此我们从添加一个有名称的分割线至fillContextMenu()方法（参见7.3.2节）开始。

```
menuMgr.add(new Separator("edit"));
```

然后，我们在插件清单中声明一个新命令（参见6.1.1节）、一个新的menuContribution（参见6.2.6节）和一个新的处理器（参见6.3节）。使用以下的locatorURI以使复制命令仅在Favorites视图上下文菜单的正确位置出现。

```
popup:com.qualityeclipse.favorites.views.FavoritesView?before=edit
```

在Favorites视图上下文菜单中，Copy命令即使是没有内容被选中也是可用的。为了解决这一问

题, 添加该enabledWhen表达式至处理器以使菜单项仅当在Favorites视图中有一个或多个项被选中时才可用。

```
<handler class=
    "com.qualityeclipse.favorites.handlers.CopyFavoritesHandler"
    commandId="com.qualityeclipse.favorites.commands.copy">
    <enabledWhen>
        <with variable="selection">
            <count value="+"/>
        </with>
    </enabledWhen>
</handler>
```

3. 剪切

剪切处理器是基于复制和移除处理器。首先使用复制处理器以复制选中的Favorites项至剪贴板, 然后使用移除处理器以从Favorites视图移除选中项。它和上一节描述的复制处理器的初始化和使用方式十分类似。

```
public class CutFavoritesHandler extends AbstractHandler
{
    IHandler copy = new CopyFavoritesHandler();
    IHandler remove = new RemoveFavoritesHandler();
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        copy.execute(event);
        remove.execute(event);
        return null;
    }
    public void dispose() {
        copy.dispose();
        remove.dispose();
        super.dispose();
    }
}
```

4. 粘贴

粘贴操作将之前被其他操作添加至剪贴板的信息添加至Favorites视图。与复制操作类似(参见7.3.7节), 传输对象降低了从平台特定的字节流传输至对象的复杂程度, 并且粘贴操作将这些对象转换为添加至Favorites视图的项。将execute(ExecutionEvent)方法提取至一个名为ClipboardHandler的新类以使它可以被CopyFavoritesHandler和新的PasteFavoritesHandler使用。粘贴操作的初始化和使用与7.3.7节中讨论的复制操作十分类似。

```
public class PasteFavoritesHandler extends ClipboardHandler
{
    protected Object execute(ExecutionEvent evt, Clipboard clipboard)
        throws ExecutionException {
        if (!paste(clipboard, JavaUI.getJavaElementClipboardTransfer()))
            paste(clipboard, ResourceTransfer.getInstance());
        return null;
    }
    private void paste(Clipboard clipboard, Transfer transfer) {
        Object[] elements = (Object[]) clipboard.getContents(transfer);
        if (elements != null && elements.length != 0) {
            FavoritesManager.getManager().addFavorites(elements);
        }
    }
}
```

```

        return true;
    }
    return false;
}
}

```

7.3.8 拖放支持

我们可以使用复制/粘贴操作来从另一个视图添加对象至Favorites视图,但如果可以允许对象拖进拖出Favorites视图那将是十分理想的。为了完成这项任务,通过调用下列来源于createPartControl()的新方法添加拖动源和放目标对象至Favorites视图。FavoritesDragSource和FavoritesDropTarget类型在之后的两节定义。

```

private void hookDragAndDrop() {
    new FavoritesDragSource(viewer);
    new FavoritesDropTarget(viewer);
}

```

1. 将对象拖出收藏夹视图

FavoritesDragSource类型初始化拖动源操作并处理Favorites项至资源对象和文本的转换。这些功能允许用户拖放选中Favorites项至Eclipse其他任何地方或其他支持拖放的程序,如Microsoft Word(图7-7)。

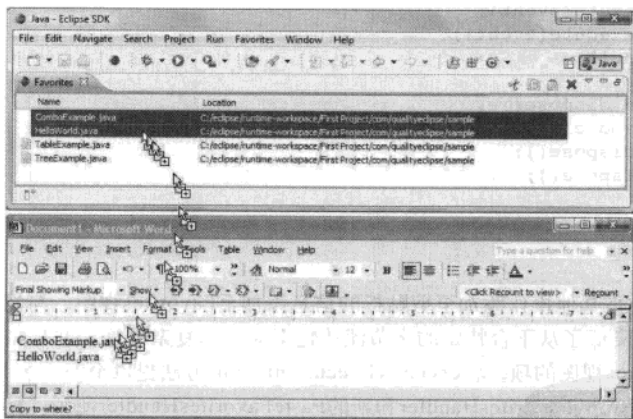


图7-7 从Microsoft Word至Eclipse的拖放操作

从hookDragAndDrop()方法(参见7.3.8节)调用的构造函数,通过以下方法初始化拖动源:

- 创建拖动源——new DragSource()
- 指定可用操作——DND.DROP_COPY

如果项可以被移动和复制,可以在DND.DROP_MOVE | DND.DROP_COPY指定多个操作。

- 指定可用的数据类型——资源和文本(为了了解更多信息,参见7.3.8节)。
- 将其自身作为DragSourceListener添加以处理从Favorites项至资源或文本的数据转换。

当用户从Favorites视图初始化拖动操作时,将调用dragStart()方法以确定可以执行拖动操作。在

这种情况中,如果有Favorites项被选中,设置event.doit字段为true。否则,因为当有至少一个Favorites项被选中时才能执行操作,将event.doit字段为false。当用户放置对象时,将调用dragSetData()方法以在传输发生之前转换选中项,然后在传输完成之后,调用dragFinish()方法。

```
public class FavoritesDragSource
    implements DragSourceListener
{
    private final TableViewer viewer;

    public FavoritesDragSource(TableViewer viewer) {
        this.viewer = viewer;
        DragSource source =
            new DragSource(viewer.getControl(), DND.DROP_COPY);
        source.setTransfer(new Transfer[] {
            TextTransfer.getInstance(),
            ResourceTransfer.getInstance() });
        source.addDragListener(this);
    }

    public void dragStart(DragSourceEvent event) {
        event.doit = !viewer.getSelection().isEmpty();
    }

    public void dragSetData(DragSourceEvent event) {
        Object[] objects =
            ((IStructuredSelection) viewer.getSelection()).toArray();
        if (ResourceTransfer.getInstance().isSupportedType(
            event.dataType)) {
            event.data = CopyFavoritesHandler.asResources(objects);
        }
        else if (TextTransfer.getInstance().isSupportedType(
            event.dataType)) {
            event.data = CopyFavoritesHandler.asText(objects);
        }
    }

    public void dragFinished(DragSourceEvent event) {
        // If this was a MOVE operation,
        // then remove the items that were moved.
    }
}
```

2. 拖入对象至收藏夹视图

FavoritesDropTarget类型允许从另一个视图拖动项添加至Favorites视图。这项功能允许用户从Resource Navigator视图或Java Package视图拖动资源或Java元素至Favorites视图。

从hookDragAndDrop()方法(7.3.8节,拖放支持)调用的构造函数通过以下方法初始化放置目标:

- 创建放置目标——new DropTarget()
- 指定被接受的操作——DND.DROP_MOVE | DND.DROP_COPY
为了便利,指定允许移动操作。但当执行实际操作时,将其转换为复制操作。
- 指定接受的数据类型——资源和Java元素
(为了了解更多信息,参见7.3.8节。)
- 将它自身作为DropTargetListener添加,以处理从对象至Favorites项的数据转换。

在拖动操作期间, 将有几个事件发生以使不同的放置目标可以提供反馈给用户。包括当指针进入时、移动经过时和离开放置目标时。当你需要添加项至Favorites视图而不从它们的原始位置移除时, 并让用户能方便地进行操作, 以使不必按下Ctrl键以执行拖动操作, 实现dragEnter()方法以将移动操作转换为复制操作。从移动操作至复制操作的转换完成于dragEnter()方法和drop()方法以使用户获得表示复制将会在执行操作之前发生的视觉反馈。

当用户在Favorites视图中放置对象时, 将调用drop()方法以执行该操作。它将对象转换为Favorites项并保证操作确实是一个复制操作以使对象不会从它们的原始位置移除。

```
public class FavoritesDropTarget extends DropTargetAdapter
{
    public FavoritesDropTarget(TableViewer viewer) {
        DropTarget target =
            new DropTarget(viewer.getControl(), DND.DROP_MOVE
                | DND.DROP_COPY);
        target.setTransfer(new Transfer[] {
            ResourceTransfer.getInstance(),
            JavaUI.getJavaElementClipboardTransfer() });
        target.addDropListener(this);
    }

    public void dragEnter(DropTargetEvent event) {
        if (event.detail == DND.DROP_MOVE
            || event.detail == DND.DROP_DEFAULT) {
            if ((event.operations & DND.DROP_COPY) != 0)
                event.detail = DND.DROP_COPY;
            else
                event.detail = DND.DROP_NONE;
        }
    }

    public void drop(DropTargetEvent event) {
        FavoritesManager manager = FavoritesManager.getManager();
        if (JavaUI.getJavaElementClipboardTransfer().isSupportedType(
            event.currentDataType)
            && (event.data instanceof IJavaElement[])) {
            manager.addFavorites((IJavaElement[]) event.data);
            event.detail = DND.DROP_COPY;
        }
        else if (ResourceTransfer.getInstance().isSupportedType(
            event.currentDataType)
            && (event.data instanceof IResource[])) {
            manager.addFavorites((IResource[]) event.data);
            event.detail = DND.DROP_COPY;
        }
        else
            event.detail = DND.DROP_NONE;
    }
}
```

3. 自定义传输类型

传输对象将不同的格式, 如资源, 转换为平台特定的字节流并实行反向转换, 这样使得信息可以在不同程序间交换。Eclipse提供几种传输类型, 包括:

- ByteArrayTransfer
- EditorInputTransfer

- FileTransfer
- JavaElementTransfer
- MarkerTransfer
- PluginTransfer
- ResourceTransfer
- RTFTransfer
- TextTransfer

这些传输对象对于普通类型的对象，如资源，是十分有用的。如果你拖动程序特定的对象从一个视图到另一个视图，然而被传输的对象可能不会完全获取被拖动对象的所有信息。比如，你想要从Favorites视图拖动Favorites项至另一个视图，而该项具有附加状态属性，同时还使用了ResourceTransfer对象，那么附加状态信息将会丢失。

解决这一问题需要创建一个自定义传输类型，比如以下内容中所示。传输类型必须是org.eclipse.swt.dnd.Transfer类的子类。但是，继承org.eclipse.swt.dnd.ByteArrayTransfer是更容易的。这是由于它提供了额外的行为。如果为Favorites项创建了自定义传输类型，那么它将依赖于7.5.2节介绍的功能，并且可能与已有的ResourceTransfer类型相似。

```
package com.qualityeclipse.favorites.views;

import ...;

public class FavoritesTransfer extends ByteArrayTransfer
{
    private static final FavoritesTransfer INSTANCE =
        new FavoritesTransfer();
    public static FavoritesTransfer getInstance() {
        return INSTANCE;
    }

    private FavoritesTransfer() {
        super();
    }
}
```

每一个FavoritesTransfer类必须具有一个唯一的标识符以保证不同的Eclipse程序使用不同“类型”的FavoritesTransfer类。getTypeIds()和getTypeNames()方法返回平台特定的ID和可以使用该传输代理进行转换的数据类型的名称。

```
private static final String TYPE_NAME =
    "favorites-transfer-format:"
        + System.currentTimeMillis()
        + ":"
        + INSTANCE.hashCode();
private static final int TYPEID =
    registerType(TYPE_NAME);

protected int[] getTypeIds() {
    return new int[] { TYPEID };
}

protected String[] getTypeNames() {
    return new String[] { TYPE_NAME };
}
```



javaToNative()方法将Java数据表示转换为平台特定的数据表示,然后通过将其放置于TransferData参数中返回该信息。

```
protected void javaToNative(
    Object data,
    TransferData transferData) {

    if (!(data instanceof IFavoriteItem[])) return;
    IFavoriteItem[] items = (IFavoriteItem[]) data;

    /**
     * The serialization format is:
     * (int) number of items
     * Then, the following for each item:
     * (String) the type of item
     * (String) the item-specific info glob
     */
    try {
        ByteArrayOutputStream out =
            new ByteArrayOutputStream();
        DataOutputStream dataOut =
            new DataOutputStream(out);
        dataOut.writeInt(items.length);
        for (int i = 0; i < items.length; i++) {
            IFavoriteItem item = items[i];
            dataOut.writeUTF(item.getType().getId());
            dataOut.writeUTF(item.getInfo());
        }
        dataOut.close();
        out.close();
        super.javaToNative(out.toByteArray(), transferData);
    }
    catch (IOException e) {
        // Send nothing if there were problems.
    }
}
```

nativeToJava()方法将平台相关的数据表示转换为Java表示。

```
protected Object nativeToJava(TransferData transferData) {
    /**
     * The serialization format is:
     * (int) number of items
     * Then, the following for each item:
     * (String) the type of item
     * (String) the item-specific info glob
     */
    byte[] bytes =
        (byte[]) super.nativeToJava(transferData);
    if (bytes == null)
        return null;
    DataInputStream in =
        new DataInputStream(
            new ByteArrayInputStream(bytes));
    try {
        FavoritesManager mgr =
            FavoritesManager.getManager();
    }
```



```

    int count = in.readInt();
    List<IFavoriteItem> items =
        new ArrayList<IFavoriteItem>(count);
    for (int i = 0; i < count; i++) {
        String typeId = in.readUTF();
        String info = in.readUTF();
        items.add(mgr.newFavoriteFor(typeId, info));
    }
    return items.toArray(new IFavoriteItem[items.size()]);
}
catch (IOException e) {
    return null;
}
}

```

提示 在和上面类似的输入/输出 (I/O) 代码中, 可以考虑在 `ByteArrayOutputStream` 和 `DataOutputStream` 之间使用 `BufferedOutputStream`。虽然它不是一直都必需的, 但它是一种有效提高性能的方法。

7.3.9 内联编辑

另一项你需要具有的功能是在 Favorites 视图中快速简易地直接编辑 Favorites 项名称的功能。可以论证的是, 它应当触发重命名处理器或重构以使底层资源或 Java 元素将会被重命名而不是仅仅编辑项自身的名称, 但是为了阐述内联编辑功能, 我们力图简洁。

要为 Favorites 项的名称执行内联操作, 一个名为 `RenameFavoriteHandler` 的新处理器是必需的。当用户选择上下文菜单中的 `Rename` 命令时, 将打开一个包含 Favorites 视图的选中项的名称的文本字段 (图 7-8)。用户在该文本字段中输入新的名称, 然后按下返回键, 这将关闭编辑器并更新项的名称。

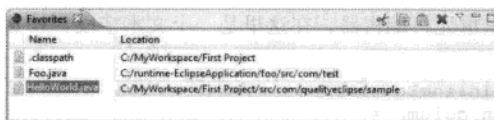


图 7-8 显示内联文本字段的收藏夹视图

该新处理器获取当前选择并打开一个单元格编辑器以编辑第一个选中元素的名称。

```

public class RenameFavoritesHandler extends AbstractHandler
{
    private static final int COLUMN_TO_EDIT = 1;

    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        IWorkbenchPart part = HandlerUtil.getActivePart(event);
        if (!(part instanceof FavoritesView))
            return null;
        editElement((FavoritesView) part);
        return null;
    }

    public void editElement(FavoritesView favoritesView) {
        TableViewer viewer =
            favoritesView.getFavoritesViewer();
    }
}

```



```

IStructuredSelection selection =
    (IStructuredSelection) viewer.getSelection();
if (!selection.isEmpty())
    viewer.editElement(
        selection.getFirstElement(), COLUMN_TO_EDIT);
}
}

```

为了使Rename...命令出现于上下文菜单中，我们在插件清单中声明一个新command（参见6.1.1节）、一个新的menuContribution（参见6.2.6节）和相应的handler（参见6.3节）。使用以下的locatorURI以使Rename...命令仅出现于Favorites视图上下文菜单的正确位置。

```
popUp:com.qualityeclipse.favorites.views.FavoritesView?before=edit
```

在Favorites视图上下文菜单中，Rename...命令当没有东西被选中时也可用。为了解决该问题，添加该enabledWhen表达式至处理器以使菜单项仅当Favorites视图中有至少一项被选中时才可用。

```

<handler class=
    "com.qualityeclipse.favorites.handlers.RenameFavoritesHandler"
    commandId="com.qualityeclipse.favorites.commands.rename">
    <enabledWhen>
        <with variable="selection">
            <count value="+"/>
        </with>
    </enabledWhen>
</handler>

```

接下来，我们必须修改FavoritesView表以当选择Rename...命令时打开一个文本单元格编辑器，在该单元格中显示合适的文本，并存储修改过的值至底层模型。我们首先修改createPartControl以调用新的createInlineEditor方法。该方法初始化一个新的TableViewerColumn。该TableViewerColumn负责调正单元格位于被重命名的项的位置和大小，并管理该单元格编辑器的生命周期。EditingSupport对象初始化该单元格编辑器，在这里是一个文本字段，初始化它的值并存储所生成的用户修改至收藏夹模型。

```

private void createInlineEditor() {
    TableViewerColumn column =
        new TableViewerColumn(viewer, nameColumn);

    column.setLabelProvider(new ColumnLabelProvider() {
        public String getText(Object element) {
            return ((IFavoriteItem) element).getName();
        }
    });

    column.setEditingSupport(new EditingSupport(viewer) {
        TextCellEditor editor = null;

        protected boolean canEdit(Object element) {
            return true;
        }

        protected CellEditor getCellEditor(Object element) {
            if (editor == null) {
                Composite table = (Composite) viewer.getControl();
                editor = new TextCellEditor(table);
            }
        }
    });
}

```

```

        return editor;
    }
    protected Object getValue(Object element) {
        return ((IFavoriteItem) element).getName();
    }

    protected void setValue(Object element, Object value) {
        ((IFavoriteItem) element).setName((String) value);
        viewer.refresh(element);
    }
    });
}

```

此时，用户可以在上下文菜单中选择Rename...命令或在Favorites视图中点击名称以命名一个特定的收藏夹项。TableViewerColumn默认提供了“点击重命名”行为，而这种行为与我们想要的不是十分符合。要修改该行为，我们添加以下内容至createInlineEditor方法的末尾以允许单元格编辑仅当它在程序中被触发，比如从RenameFavoritesHandler，或当用户按下Alt并点击Favorites视图中的名称以触发。

```

viewer.getColumnViewerEditor().addEditorActivationListener(
    new ColumnViewerEditorActivationListener() {

        public void beforeEditorActivated(
            ColumnViewerEditorActivationEvent event) {
            if (event.eventType == event.MOUSE_CLICK_SELECTION) {
                if (!(event.sourceEvent instanceof MouseEvent))
                    event.cancel = true;
                else {
                    MouseEvent m = (MouseEvent) event.sourceEvent;
                    if ((m.stateMask & SWT.ALT) == 0)
                        event.cancel = true;
                }
            }
            else if (event.eventType != event.PROGRAMMATIC)
                event.cancel = true;
        }

        public void afterEditorActivated(
            ColumnViewerEditorActivationEvent event) {
        }

        public void beforeEditorDeactivated(
            ColumnViewerEditorDeactivationEvent event) {
        }

        public void afterEditorDeactivated(
            ColumnViewerEditorDeactivationEvent event) {
        }
    });

```

我们还需要关联该方法以使用户可以按下F2以直接编辑项的名称与前面将Delete键与删除操作关联的方法类似（参见7.3.5节）。添加下列内容至handleKeyReleased方法以完成这项任务。

```

if (event.keyCode == SWT.F2 && event.stateMask == 0) {
    new RenameFavoritesHandler().editElement(this);
}

```

7.4 链接视图

在许多情况下，活动视图的当前选择可以影响其他视图中的选择，打开编辑器，改变选中编辑器，或在一个已打开的编辑器中更改选择。比如，在Java浏览透视图（参见1.2.1节）中，在Types视图中更改选择将改变Projects和Packages视图中的选择，改变Members视图中显示的内容，并更改活动编辑器。对于一个既要发布它自己的选择，又要使用活动部分的选择的视图来说，它必须同时为选择提供者（selection provider）和选择监听者（selection listener）。

7.4.1 选择提供者

视图要成为选择提供者，它必须在视图站点中注册它自身为选择提供者。此外，该视图包含的所有对象必须是可适应的（参见下一节）以使其他对象可以将选中对象适配为它们可以理解的对象。在Favorites视图中，可以通过将以下内容添加至createTableViewer()方法以注册视图为选择提供者：

```
getSite().setSelectionProvider(viewer);
```

7.4.2 可适配对象

org.eclipse.core.runtime.IAdaptable接口允许对象将一种它可能无法理解的对象类型转换为另一种它可以获取信息和进行操作的对象类型（更多关于适配器的信息在21.3节）。对于Favorites视图而言，这意味着IFavoritesItem接口必须继承IAdaptable接口，而后的两个getAdapter()方法必须分别添加至FavoriteResource和FavoriteJavaElement。

```
public Object getAdapter(Class adapter) {
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager().getAdapter(this, adapter);
}

public Object getAdapter(Class adapter) {
    if (adapter.isInstance(element))
        return element;
    IResource resource = element.getResource();
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager().getAdapter(this, adapter);
}
```

7.4.3 选择监听器

对于一个要处理其他部分的选择的视图而言，它必须添加一个选择监听器至页面以使得当活动部分改变时或活动部分中的选择改变时，它可以通过恰当的改变它自身的选择进行响应。对于Favorites视图而言，如果选择包含可以被适配于视图中的对象的对象，那么视图应调整它的选择。为了完成该项任务，在createPartControl()方法的尾部添加一个对以下新hookPageSelection()方法的调用。

```
private ISelectionListener pageSelectionListener;

private void hookPageSelection() {
    pageSelectionListener = new ISelectionListener() {
        public void selectionChanged(
            IWorkbenchPart part,
            ISelection selection) {
            pageSelectionChanged(part, selection);
        }
    };
}
```

```

    }
};
getSite().getPage().addPostSelectionListener(
    pageSelectionListener);
}

protected void pageSelectionChanged(
    IWorkbenchPart part,
    ISelection selection
) {
    if (part == this)
        return;
    if (!(selection instanceof IStructuredSelection))
        return;
    IStructuredSelection sel = (IStructuredSelection) selection;
    IFavoriteItem[] items = FavoritesManager.getManager()
        .existingFavoritesFor(sel.iterator());
    if (items.length > 0)
        viewer.setSelection(new StructuredSelection(items), true);
}

```

然后, 添加以下内容至dispose()方法, 当Favorites视图被关闭时进行清理工作。

```

if (pageSelectionListener != null)
    getSite().getPage().removePostSelectionListener(
        pageSelectionListener);

```

7.4.4 打开编辑器

当用户双击Favorites视图中的文件时, 应打开一个文件编辑器。为了完成这项任务, 添加一个新FavoritesView方法。该方法从createPartControl()方法调用。

```

private void hookMouse() {
    viewer.getTable().addMouseListener(new MouseAdapter() {
        public void mouseDoubleClick(MouseEvent e) {
            EditorUtil.openEditor(getSite().getPage(),
                viewer.getSelection());
        }
    });
}

```

该方法在新EditorUtil方法中引用了一个新的静态方法。该新静态方法检查当前选择的第一个元素, 如果该元素是IFile的实例, 将在该文件中打开一个编辑器。

```

public static void openEditor(
    IWorkbenchPage page, ISelection selection)
{
    // Get the first element.

    if (!(selection instanceof IStructuredSelection))
        return;
    Iterator<?> iter = ((IStructuredSelection) selection).iterator();
    if (!iter.hasNext())
        return;
    Object elem = iter.next();
    // Adapt the first element to a file.

```

```
if (!(elem instanceof IAdaptable))
    return;

IFile file = (IFile) ((IAdaptable) elem).getAdapter(IFile.class);
if (file == null)
    return;

// Open an editor on that file.

try {
    IDE.openEditor(page, file);
}
catch (PartInitException e) {
    FavoritesLog.logError(
        "Open editor failed: " + file.toString(), e);
}
}
```

7.5 保存视图状态

到此时为止，当Eclipse会话启动时，Favorites视图仅包含当前项目的列表。可以在会话周期内添加项至Favorites视图。但当Eclipse被关闭时，更改会丢失。此外，视图的排序和过滤信息应得到保存以使视图在会话重启时返回同样的状态。要完成所有的这些任务，需要用到两种不同的机制。

7.5.1 保存本地视图信息

Eclipse提供了一种基于记忆的机制用于保存视图和编辑器状态信息。在这里，由于信息对于每个独立的视图是不同的，所以该机制对于保存视图排序和过滤程序状态是合适的。但它不适合用于保存由多个视图共享的全局信息。我们将在下一节解决该问题。

为了保存排序状态，必须添加两个方法至FavoritesViewSorter。第一个方法通过将排序顺序和升序/降序状态转换为一个类XML结构以保存当前排序状态为IMemento的实例。第二个方法采用了一种非常谨慎的方法来读取和重设来源于IMemento的排序顺序和升序/降序状态。这样，即使IMemento不是所预期的，排序状态也将是合法的。

```
private static final String TAG_DESCENDING = "descending";
private static final String TAG_COLUMN_INDEX = "columnIndex";
private static final String TAG_TYPE = "SortInfo";
private static final String TAG_TRUE = "true";

public void saveState(IMemento memento) {
    for (int i = 0; i < infos.length; i++) {
        SortInfo info = infos[i];
        IMemento mem = memento.createChild(TAG_TYPE);
        mem.putInteger(TAG_COLUMN_INDEX, info.columnIndex);
        if (info.descending)
            mem.putString(TAG_DESCENDING, TAG_TRUE);
    }
}

public void init(IMemento memento) {
    List<SortInfo> newInfos = new ArrayList<SortInfo>(infos.length);
    IMemento[] mems = memento.getChildren(TAG_TYPE);
    for (int i = 0; i < mems.length; i++) {
        IMemento mem = mems[i];
```

```
        Integer value = mem.getInteger(TAG_COLUMN_INDEX);
        if (value == null)
            continue;
        int index = value.intValue();
        if (index < 0 || index >= infos.length)
            continue;
        SortInfo info = infos[index];
        if (newInfos.contains(info))
            continue;
        info.descending =
            TAG_TRUE.equals(mem.getString(TAG_DESCENDING));
        newInfos.add(info);
    }
    for (int i = 0; i < infos.length; i++)
        if (!newInfos.contains(infos[i]))
            newInfos.add(infos[i]);
    infos = newInfos.toArray(new SortInfo[newInfos.size()]);
}
```

除了保存排序状态外，过滤器状态也需要保存。可以通过添加以下两个方法至FavoritesView FilterAction类型完成。

```
public void saveState(IMemento memento) {
    nameFilter.saveState(memento);
}

public void init(IMemento memento) {
    nameFilter.init(memento);
}
```

然后添加两个新方法至FavoritesViewNameFilter:

```
private static final String TAG_PATTERN = "pattern";
private static final String TAG_TYPE = "NameFilterInfo";

public void saveState(IMemento memento) {
    if (pattern.length() == 0)
        return;
    IMemento mem = memento.createChild(TAG_TYPE);
    mem.putString(TAG_PATTERN, pattern);
}

public void init(IMemento memento) {
    IMemento mem = memento.getChild(TAG_TYPE);
    if (mem == null)
        return;
    setPattern(mem.getString(TAG_PATTERN));
}
```

这些新方法将通过添加以下字段和方法至FavoritesView与视图关联。

```
private IMemento memento;

public void saveState(IMemento memento) {
    super.saveState(memento);
    sorter.saveState(memento);
    filterAction.saveState(memento);
}
```



```

public void init(IViewSite site, IMemento memento)
    throws PartInitException
{
    super.init(site, memento);
    this.memento = memento;
}

```

排序和过滤状态还不能立即存储在上面所示的init()方法中，这是因为还没有创建组件控件。作为替代，方法缓存IMemento以在初始化过程中使用。然后，在将排序程序与查看器关联，将过滤操作与菜单关联之前，你必须修改临近存储排序和过滤状态的createTableSorter()方法和createViewPulldownMenu()方法。

```

private void createTableSorter() {

    ... same code as in Section 7.2.6 on page 308 ...
    if (memento != null)
        sorter.init(memento);
    viewer.setSorter(sorter);
}

private void createViewPulldownMenu() {

    ... same code as in a Section 7.3.4 on page 319 ...

    if (memento != null)
        filterAction.init(memento);
    menu.add(filterAction);
}

```

Eclipse存储所有基于记忆的视图和编辑器状态信息于一个文件中：

<workspace>\.metadata\.plugins\org.eclipse.ui.workbench\workbench.xml

例如（经过重新格式化以更容易阅读）：

```

<views>
  <view
    id="com.qualityeclipse.favorites.views.FavoritesView"
    partName="Favorites">
    <viewState>
      <SortInfo columnIndex="0" descending="true"/>
      <SortInfo columnIndex="1"/>
      <SortInfo columnIndex="2"/>
    </viewState>
  </view>
  <view id="org.eclipse.ui.views.TaskList" partName="Tasks">
    <viewState
      columnWidth0="19" columnWidth1="19" columnWidth2="288"
      columnWidth3="108" columnWidth4="216" columnWidth5="86"
      horizontalPosition="0" verticalPosition="0">
      <selection/>
    </viewState>
  </view>
  ...
</views>

```

7.5.2 保存全局视图信息

现在，你需要保存FavoritesManager的状态。它由所有Favorites视图共享。要实现这一功能，扩

展FavoritesActivator、FavoritesManager和所有的Favorites项，以使它们具有保存他们信息的功能。这样使得他们可以在稍后被重新创建。在FavoritesActivator中，扩展stop()方法以在FavoritesManager中调用新的saveFavorites()方法。

```
FavoritesManager.getManager().saveFavorites();
```

FavoritesManager中已有的loadFavorites()方法必须根据以下内容进行修改并添加新方法，以使Favorites项当需要时可以被惰性载入。惰性初始化是Eclipse的主题，所以列表只有当需要它时才会被创建。此外，还必须添加一个新saveFavorites()方法以存储收藏夹项。这样当Eclipse重启时，它们可以被存储。

```
private static final String TAG_FAVORITES = "Favorites";
private static final String TAG_FAVORITE = "Favorite";
private static final String TAG_TYPEID = "TypeId";
private static final String TAG_INFO = "Info";
private void loadFavorites() {
    favorites = new HashSet<IFavoriteItem>(20);
    FileReader reader = null;
    try {
        reader = new FileReader (getFavoritesFile());
        loadFavorites(XMLMemento.createReadRoot(reader));
    }
    catch (FileNotFoundException e) {
        // Ignored... no Favorites items exist yet.
    }
    catch (Exception e) {
        // Log the exception and move on.
        FavoritesLog.logError(e);
    }
    finally {
        try {
            if (reader != null) reader.close();
        } catch (IOException e) {
            FavoritesLog.logError(e);
        }
    }
}

private void loadFavorites(XMLMemento memento) {
    IMemento [] children = memento.getChildren(TAG_FAVORITE);
    for (int i = 0; i < children.length; i++) {
        IFavoriteItem item =
            newFavoriteFor(
                children[i].getString(TAG_TYPEID),
                children[i].getString(TAG_INFO));
        if (item != null)
            favorites.add(item);
    }
}

public IFavoriteItem newFavoriteFor(String typeId, String info) {
    FavoriteItemType[] types = FavoriteItemType.getTypes();
    for (int i = 0; i < types.length; i++)
        if (types[i].getId().equals(typeId))
            return types[i].loadFavorite(info);
    return null;
}
```



```

    }

    public void saveFavorites() {
        if (favorites == null)
            return;
        XMLMemento memento = XMLMemento.createWriteRoot(TAG_FAVORITES);
        saveFavorites(memento);
        FileWriter writer = null;
        try {
            writer = new FileWriter(getFavoritesFile());
            memento.save(writer);
        }
        catch (IOException e) {
            FavoritesLog.logError(e);
        }
        finally {
            try {
                if (writer != null)
                    writer.close();
            }
            catch (IOException e) {
                FavoritesLog.logError(e);
            }
        }
    }

    private void saveFavorites(XMLMemento memento) {
        Iterator<IFavoriteItem> iter = favorites.iterator();
        while (iter.hasNext()) {
            IFavoriteItem item = (IFavoriteItem) iter.next();
            IMemento child = memento.createChild(TAG_FAVORITE);
            child.putString(TAG_TYPEID, item.getType().getId());
            child.putString(TAG_INFO, item.getInfo());
        }
    }

    private File getFavoritesFile() {
        return FavoritesActivator
            .getDefault()
            .getStateLocation()
            .append("favorites.xml")
            .toFile();
    }
}

```

load和save方法与一个名为favorites.xml的文件进行交互。该文件位于下列工作区元数据子目录中: <workspace>\.metadata\plugins\com.qualityeclipse.favorites。该文件内容是XML格式的, 而且可能与下列代码类似:

```

<?xml version="1.0" encoding="UTF-8"?>
<Favorites>
    <Favorite
        Info="/First Project/com/qualityeclipse/sample
        /HelloWorld.java"
        TypeId="WBFile"/>
    <Favorite
        Info="/com.qualityeclipse.favorites/src"
        TypeId="WBFolder"/>
    ...
</Favorites>

```

提示 Eclipse可能会崩溃或锁死。如果它崩溃了，那么正常关闭顺序将会优先执行，这样，你的插件将不会得到保存它的模型状态的机会。要保护你的数据，你可以注册一个保存参与者（ISaveParticipant）并在Eclipse会话期内在不同时期保存关键模型状态（“快照”）。这种机制与当你的插件是不活动时接受资源变更事件中所使用的机制是一样的（参见9.5节）。

7.6 测试

当Favorites视图已经被修改完毕，Favorites视图的JUnit测试需要更新以响应这些修改。如果测试不经过修改继续运行，你将得到以下失败信息。

```
testView(com.qualityeclipse.favorites.test.FavoritesViewTest)

java.lang.AssertionError: array lengths differed, expected.length=3
actual.length=0
    at org.junit.Assert.fail(Assert.java:71)
    at org.junit.Assert.internalArrayEquals(Assert.java:293)
    at org.junit.Assert.assertArrayEquals(Assert.java:129)
    at org.junit.Assert.assertArrayEquals(Assert.java:140)
    at com.qualityeclipse.favorites.test.
        FavoritesViewTest.testView(FavoritesViewTest.java:63)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    ... etc ...
```

根据进一步的观察，该测试在寻找默认的查看器内容（参见2.8.3节）。当该默认内容根据实际内容的需要被移除（参见7.2.4节）后，测试应根据以下内容进行修改：

```
public void testView() {
    TableViewer viewer = testView.getFavoritesViewer();

    Object[] expectedContent = new Object[] { };
    Object[] expectedLabels = new String[] { };

    ... code for the rest of the test ...
}
```

以一种同样的方式，添加代码至AddToFavoritesTest（参见6.7.6节）以在测试之前和之后维护Favorites视图内容。由于这种类型的维护将被复制至几个地方，它可以被释放至一个新的assertFavoritesViewContent方法，并提升至AbstractFavoritesTest类。

提示 添加org.junit.Assert至Java > Editor > Content Assist > Favorites首选项页，以使Ctrl-空格辅助建议将包含不同的JUnit Assert静态方法。

7.7 图像缓存

Image是一种Java基础元素。它包装本地资源，因此必须得到良好的管理。与Eclipse中其他所有本地包装者类似，原则是：谁创建，谁销毁。以防止发生内存泄漏。另一方面，ImageDescriptor是一个纯正的Java类型。它定义了特殊图像，但没有包含它的相关本地资源。它不需要进行正确的管理和移除。它甚至会被Java垃圾收集器自动管理和释放。

当插件创建了一个Image实例，它一般都会将其缓存至一个为映射该图像标识符至特定图像的对

象 (一般是一个ImageDescriptor)。缓存不仅记忆哪一个Image实例被创建并因而需要被清理, 它还保存了同一个图像使得不需要多次读入内存, 防止对操作系统资源不必要的使用。取决于何处和何时使用图像, 图像缓存在视图关闭时可能会被释放, 或者它在插件的整个生命周期中将得到保存。

在Favorites插件中, 如果你需要载入你自己的图像 (参见7.2.3节), 实例化一个与下面所示类似的类以缓载体入的图像。该类通过惰性载入图像遵循了Eclipse的原则。它当图像被请求时才载入图像, 而不是当插件启动时或视图第一次打开时就立即载入所有图像。插件的stop()方法将会被修改以调用该实例的dispose()方法, 以使图像在插件被关闭时可以得到清理。

```
public class ImageCache {
    private final Map<ImageDescriptor, Image> imageMap =
        new HashMap<ImageDescriptor, Image>();
    public Image getImage(ImageDescriptor imageDescriptor) {
        if (imageDescriptor == null)
            return null;
        Image image = (Image) imageMap.get(imageDescriptor);
        if (image == null) {
            image = imageDescriptor.createImage();
            imageMap.put(imageDescriptor, image);
        }
        return image;
    }

    public void dispose() {
        Iterator<Image> iter = imageMap.values().iterator();
        while (iter.hasNext())
            iter.next().dispose();
        imageMap.clear();
    }
}
```

提示 WindowBuilder Pro (附录A) 提供了ResourceManager用于缓存图像、字体和光标等。

7.8 自动调整大小的表列

另一项对于Favorites视图的良好改进是表中的列可以自动调正大小以适应当前的空间。Eclipse为自动调整大小的表提供了TableColumnLayout, 为自动调整大小的树提供了TreeColumnLayout。根据下列所示内容修改createView()以替代Favorites视图表布局。

```
TableColumnLayout layout =new TableColumnLayout();
parent.setLayout(layout);

typeColumn = new TableColumn(table, SWT.LEFT);
typeColumn.setText("");
layout.setColumnData(typeColumn,new ColumnPixelData(18));

nameColumn = new TableColumn(table, SWT.LEFT);
nameColumn.setText("Name");
layout.setColumnData(nameColumn,new ColumnWeightData(4));

locationColumn = new TableColumn(table, SWT.LEFT);
locationColumn.setText("Location");
layout.setColumnData(locationColumn,new ColumnWeightData(9));
```

7.9 RFRS相关事项

《RFRS Requirements》中的“用户界面”一节包含了七个关于视图的内容（五个要求和两个最佳做法）。它们都来源于Eclipse UI准则。

7.9.1 用于导航的视图 (RFRS 3.5.15)

用户界面准则#7.1是一个要求。它说明：

使用视图以在信息层次结构中导航，打开编辑器或显示对象的属性。

为了通过这项测试，创建一个由你程序定义的视图的列表并说明它们是如何用于导航信息，打开编辑器或显示某些对象的属性的。在本章前面展示的示例中，显示Favorites视图（图10-4），并向评审人描述它的作用。特别是，在Favorites视图中的文件上双击将在编辑器中打开该文件。

7.9.2 视图立即保存 (RFRS 3.5.16)

用户界面准则#7.2是一个要求，它说明：

在视图中做出的更改必须立即保存。比如，如果在Navigator中修改了一个文件，更改必须立即提交至工作区。在Outline视图中做出的更改必须立即提交至活动编辑器的编辑模型。对于在Properties视图中做出的修改，如果该属性是一个打开编辑模型的属性，它应被同步至编辑模型。如果它是文件的属性，那么同步至该文件。在过去，一些视图已经尝试过使用一个保存操作实现编辑器样式的生命周期。这将导致混淆。工作区插件的File菜单包含了保存操作，但它仅适用于活动编辑器。它不会聚焦于活动视图。这将导致File > Save操作与视图中的保存操作发生冲突的局面。

对于这项测试，显示在你的视图中，更改是如何立即得到保存的。如果你的视图更新了一个已有的编辑器，保证编辑器将立即被标记为脏的，并显示修改表示符(*)。另外，显示保存菜单不需要被激活以用于视图保存它的更改。

7.9.3 视图初始化 (RFRS 3.5.17)

用户界面准则#7.8是一个要求，它说明：

当第一次打开一个视图时，从透视图状态获取视图的输入。视图可能查询透视图的输入、选择或另一个视图的状态。比如，如果Outline视图被打开，它将决定活动编辑器，从编辑器查询大纲模型，并显示大纲模型。

为了通过这项测试，展示你的视图反映了透视图的输入状态（如果合适）。如果你的视图打算显示选中编辑器的一些属性，那么保证当该编辑器打开时，它显示恰当的信息。对于Favorites视图而言，该要求可能不适用。Favorites视图可以被扩展以更新它自己的选择以反映当前的活动编辑器。

7.9.4 视图全局操作 (RFRS 3.5.18)

用户界面准则#7.9是一个要求，它说明：

如果一个视图支持cut、copy、paste或任意全局操作，同样的操作必须对于窗口菜单或工具栏中的同样操作是可扩展的。窗口菜单包含了一些全局操作，如Edit菜单中的cut、copy、paste。如果这些操作在一个视图被支持，该视图应当将这些窗口操作关联起来以窗口菜单或工具栏中的选择产生与视图中同样操作的相同结果。这些是受支持的全局操作：undo、redo、cut、copy、paste、print、delete、find、select all和bookmark。

对于该要求，如果你的视图实现了全局操作列表中的任意项，展示这些命令也可以在窗口菜单

和工具栏中触发。对于Favorites视图来说,显示Cut、Copy、Paste和Delete (Remove) 命令可以由平台Edit菜单触发。

7.9.5 保存视图状态 (RFRS 3.5.19)

用户界面准则#7.20是一个要求,它说明:

在会话间保存每个视图的状态。如果一个视图是自启动的,并且它的输入不是来源于其他部分的选择,该视图的状态应在会话间得到保存。在工作区内部,Navigator视图的状态,包括输入和扩展状态,在会话间得到了保存。

说明你的视图在会话间保存了它的状态。对于Favorites视图来说,关闭并重启工作台,然后显示列表中的Favorites项与关闭前的工作台中的项是一样的。

7.9.6 注册上下文菜单 (RFRS 5.3.5.8)

用户界面准则#7.17是一个最佳做法,它说明:

向Eclipse注册所有视图中的上下文菜单。在Eclipse中,视图的菜单和工具栏自动被Eclipse扩展。相反,上下文菜单扩展项受视图和Eclipse联合的支持。要达到该联合,视图必须向Eclipse注册它包含的所有上下文菜单。

显示你的视图的上下文菜单可以被Eclipse扩展。如果Eclipse定义的命令适用于你视图包含的对象。这些命令应当出现于视图的上下文菜单中。对于Favorites视图来说,显示通用Eclipse命令,比如“替换”和“比较”,将在你右键点击一个Favorites项时出现(图7-9)。

7.9.7 视图操作过滤程序 (RFRS 5.3.5.9)

用户界面准则#7.18是一个最佳做法,它说明:

为视图的每一种对象类型实现一个操作过滤程序。操作过滤程序使得插件向另一个插件定义的视图中的对象添加操作变得更容易。操作目标使用对象类型和属性进行描述。

与上一个最佳做法类似,显示所有添加于你视图上下文菜单的命令适用于选中对象的类型。不适用的命令将会被过滤。对于Favorites视图来说,显示添加于上下文菜单的平台命令是基于选中对象类型上下文敏感的(图7-10)。

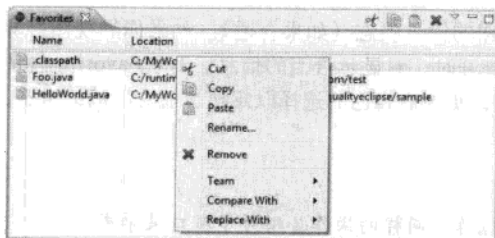


图7-9 显示了Eclipse对于上下文菜单的添加项的收藏夹视图

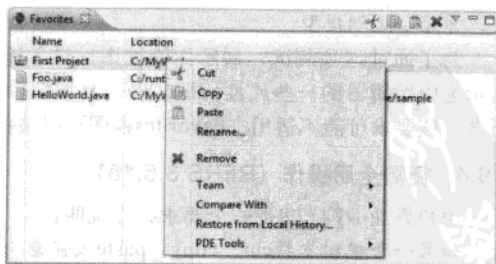


图7-10 显示了上下文菜单项基于它们的类型进行了过滤(项目显示项而不是文件)的收藏夹视图

7.10 总结

本章包含了创建新视图，修改视图以响应活动编辑器或其他视图中的选择，和输出视图选择至Eclipse其他部分的内容。下一章将讨论编辑器。它用于编辑独立资源的状态。

参考文献

本书资源 (2.9节).

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer's Guide to Eclipse, Second Edition*. Addison-Wesley, Boston, 2004.

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, Boston, 2005.

Springgay, Dave, "Creating an Eclipse View," OTI, November 2, 2001 (www.eclipse.org/articles/viewArticle/ViewArticle2.html).

Liotta, Matt, "Extending Eclipse with Helpful Views," July 20, 2004 (www.devx.com/opensource/Article/21562).



第8章 编辑器

编辑器是用户用于创建和修改资源（如文件）的主要机制。Eclipse提供了一些基本的编辑器，如文本和Java源代码编辑器。它还提供了一些更复杂的多页面编辑器，如插件清单编辑器。那些需要使用自己的编辑器的产品可以使用由Eclipse内建编辑器使用的扩展点。本章讨论创建一个新的Properties编辑器，关联相应的命令，并将其链接至Outline视图。

编辑器必须实现org.eclipse.ui.IEditorPart接口。一般地，编辑器是org.eclipse.ui.part.EditorPart的子类，因此也是org.eclipse.ui.part.WorkbenchPart的间接子类。它们从org.eclipse.ui.part.WorkbenchPart继承了大部分实现IEditorPart接口所必需的行为（图8-1）。

编辑器包含在一个org.eclipse.ui.IEditorSite中。而该org.eclipse.ui.IEditorSite又包含在org.eclipse.ui.IWorkbenchPage中。根据惰性初始化的原则，IWorkbenchPage保持org.eclipse.ui.IEditorReference的实例，而不是编辑器自身，这样使得编辑器可以不需要实际载入编辑器定义的插件而被列举和引用。



图8-1 EditorPart类

编辑器通过超类org.eclipse.ui.part.WorkbenchPart和接口org.eclipse.ui.IWorkbenchPart与视图共享通用行为集，但两者之间具有一些十分重要的区别。编辑器遵循经典的打开-修改-关闭方法，而视图中执行的任意命令应立即影响工作区和底层资源的状态。

编辑器出现于Eclipse的一个区域，而视图被安排位于编辑器区域的外部。编辑器一般是基于资源的，而视图可以显示关于一个或多个资源，甚至完全与资源无关的信息，如可用内存、网络状态或构建器错误。

8.1 编辑器声明

创建一个新的编辑器包含了两个步骤：

- 在插件清单文件中定义编辑器（图8-2）。
- 创建包含代码的编辑器组成。

一种立即着手这些任务的方法是当创建插件时创建编辑器。与视图可以被创建的方法类似（参见2.2.3节）。如果插件已经存在，那么这将成为一个两个步骤的过程。

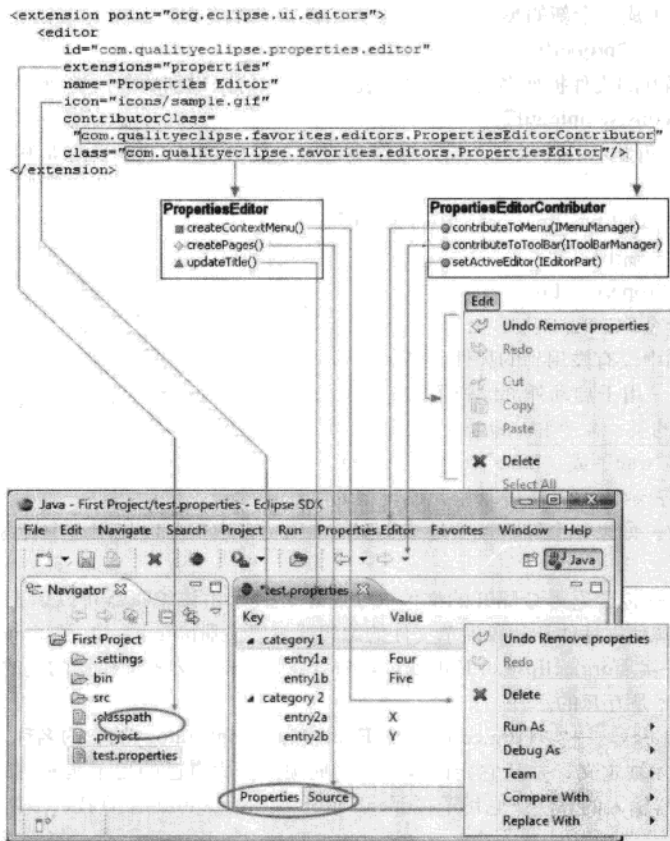


图8-2 插件清单中的编辑器声明

创建编辑器的第一步是在插件清单中定义编辑器（图8-2）。在插件清单编辑器的Extensions页面，点击右上角的Add...按钮，选择org.eclipse.ui.editors，并最后点击Finish。选择新的编辑器扩展项以在右侧显示属性，然后输入以下值。

- class——“com.qualityeclipse.favorites.editors.PropertiesEditor”

定义编辑器和实现org.eclipse.ui.IEditorPart（参见8.2节）的类的完全合格名称。点击字段右侧

的Browse...按钮以打开对话框并选择一个已有的编辑器组件。点击左侧的class标签以生成新的编辑器组件。属性的类、命令和启动器是互斥的。该类使用它的无参数构造函数进行初始化,但也可以使用IExecutableExtension接口(参见21.5节)赋予参数。

- contributorClass——“com.qualityeclipse.favorites.editors.PropertiesEditorContributor”

实现org.eclipse.ui.IEditorActionBarContributor并添加新操作至工作台菜单和工具栏,反映编辑器类型的特性(参见8.5.2节)的类的完全合格名称。该属性应在定义了类属性的情况下才被定义。点击字段右侧的Browse...按钮以打开一个对话框以用于选择一个已有的编辑器添加者。点击左侧的contributorClass以生成一个新的类。

- extensions——“properties”

一个逗号分隔开的文件扩展名的字符串,表示编辑器可以识别的文件类型。

- icon——“icons/sample.gif”

在编辑器左上角显示的图像。与操作图像类似(参见6.6.4节),该路径是相对于插件安装目录的路径。

- id——“com.qualityeclipse.properties.editor”

该编辑器的唯一标识符。

- name——“Properties Editor”

编辑器的可读名称。

其他在该示例中没有被用到的属性包括:

- command——用于启动外部编辑器的命令。该可执行的命令必须位于系统路径或在插件的目录中。属性的类、命令和启动器是互斥的。
- default——“true”或“false”(空白=false)

如果为true,表示该编辑器可以作为该类型的默认编辑器。仅当超过一个编辑器被注册用于同样的类型时才有效。如果一个编辑器不是该类型的默认值,它仍可以使用选中资源的Open with...子菜单启动。

- filenames——包含逗号分隔开的文件的字符串,表示由该编辑器可以识别的文件名。比如,一个可以理解独立插件和片段清单文件的编辑器可以注册plugin.xml、fragment.xml。
- launcher——实现org.eclipse.ui.IEditorLauncher并打开一个外部编辑器的类的名称。属性的类、命令和启动器是互斥的。
- matchingStrategy——实现org.eclipse.ui.IEditorMatchingStrategy的类的名称。该属性仅当定义了类属性时才被定义,并且它允许编辑器扩展项提供它自己的用于匹配它的编辑器之一和一个给定编辑器输入的算法。它用于在openEditor()和findEditor()中查找一个匹配的编辑器。

此外,editor元素可以具有一个或多个contentTypeBinding子元素,每一个子元素指定一个contentType。contentType引用一个org.eclipse.core.runtime.contentTypes扩展项,表示编辑器可以包含该类型的内容。contentTypes扩展项可以更精确地定义是否一个文件将关联于一个特定的编辑器,而不是只适用文件扩展名进行关联。

在根据名称和扩展名过滤了文件之后,内容类型使用一个描述器(descriptor)(IContentDescriber或ITextContentDescriber的实例)以在决定是否在文件包含特定类型的内容前扫描文件内容。Eclipse提供了几种内建的描述器,包括有:

BinarySignatureDescriber——用于二进制格式的内容描述器。二进制格式以一个已知的,固定

的偏移显示一些简单的签名。有三个参数：“签名”、“偏移量”和“是否必需”。第一个是强制的。

- **signature**——十六进制码的序列。一个对应于一个字节。比如，“CA FE BA BE”表示用于Java类文件的签名。
- **offset**——表示发现签名的第一个字节的偏移量的整数。
- **required**——表示作为IContentDescriber.INVALID或IContentDescriber.INDETERMINATE，缺少签名是否影响内容的合法性状态。

XMLRootElementContentDescriber——用于在XML文件中查找最高级元素的名称或DTD系统标识符的内容描述器。它支持两个参数：“dtd”和“element”。

8.2 编辑器组件

定义编辑器行为的代码位于实现org.eclipse.ui.IEditorPart接口的类中，一般继承以下具体类：

- org.eclipse.ui.part.EditorPart——org.eclipse.ui.IEditorPart接口的抽象基础实现。
- org.eclipse.ui.part.MultiPageEditorPart——为多页面编辑器扩展EditorPart的抽象基础实现。
- org.eclipse.ui.forms.editor.FormEditor——为多页面编辑器扩展MultiPageEditorPart的抽象基础实现。它一般使用一个或多个带有帧的页面，每一个页面用于编辑器输入的原始资源。

Properties编辑器继承了MultiPageEditorPart，并提供两个页面给用户编辑它的内容。

8.2.1 编辑器方法

以下是EditorPart的方法。

- **createPartControl(Composite)**——创建组成编辑器的控件。一般地，该方法仅仅调用更细粒度的方法，如createTree、createTextEditor等。
- **dispose()**——当编辑器被关闭并标记为编辑器生命周期的结束时，自动调用该方法。它清理所有平台资源，如图像、剪贴板等所有由该类创建的资源。该方法遵循谁创建，谁销毁的Eclipse主题。
- **doSave(IProgressMonitor)**——保存该编辑器的内容。如果保存成功，该组成将触发一个属性更改事件（PROP_DIRTY属性），以反映新的脏状态。如果保存被用户操作取消，或因为任何其他原因，组成将在IProgressMonitor上触发setCanceled以通知调用者（参见9.4节）。
- **doSaveAs()**——该方法是可选的。它打开一个Save As对话框并保存编辑器的内容至一个新位置。如果保存成功，组成将触发一个属性更改事件（PROP_DIRTY属性），以反映新的脏状态。
- **gotoMarker(IMarker)**——根据给定标记所指定的设置该编辑器的指针和选择状态。
- **init(IEditorSite, IEditorInput)**——使用给定的编辑器站点和输入初始化该编辑器。该方法在编辑器创建后的短暂时间后自动被调用。它标志着编辑器生命周期的开始。
- **isDirty()**——返回该编辑器的内容是否在上次保存操作后被更改。
- **isSaveAsAllowed()**——返回“保存为”操作是否由该组件支持。
- **setFocus()**——请求该组件在工作台范围内取得焦点。一般地，该方法在它其中一个子控件上简单地调用setFocus()。

MultiPageEditorPart提供了以下附加方法：

- **addPage(Control)**——创建并添加一个包含给定控件的新页面至该多页面编辑器。控件可以是null，允许它稍后使用setControl创建并设置。

- `addPage(IEditorPart, IEditorInput)`——创建并添加一个包含给定编辑器的新页面至该多页面编辑器。该方法还将一个属性更改监听器关联至嵌套编辑器。
- `createPages()`——为该多页面编辑器创建页面。一般地，该方法简单地调用更细粒度的方法，如`createPropertiesPage`、`createSourcePage`等。
- `getContainer()`——返回包含该多页面编辑器的页面的复杂控件。该方法应在为独立页面创建控件时作为父部件使用。也就是说，当调用`addPage(Control)`时，被传递的控件应当是该容器的后代。
- `setPageImage(int, Image)`——使用给定的索引为该页面设置图像。
- `setPageText(int, String)`——使用给定的索引为该页面设置文本标签。

8.2.2 编辑器控件

新的`PropertiesEditor`是一个包含`Properties`和`Source`页面的多页面编辑器。`Properties`页面包含了一个显示属性名/值的树，而`Source`页面显示出现于文件自身的文本。这些页面展示了根据独立控件创建编辑器（`Properties`页面）和在编辑器中嵌套另一个编辑器（`Source`页面）。

我们从创建一个`MultiPageEditorPart`的子类开始。新的`PropertiesEditor`类包含了一个`init()`方法以保证被编辑的内容类型是正确的。

```
package com.qualityclipse.favorites.editors;

import ...
import com.qualityclipse.favorites.FavoritesLog;

public class PropertiesEditor extends MultiPageEditorPart
{
    public void init(IEditorSite site, IEditorInput input)
        throws PartInitException
    {
        if (!(input instanceof IFileEditorInput))
            throw new PartInitException(
                "Invalid Input: Must be IFileEditorInput");
        super.init(site, input);
    }
}
```

然后，添加两个字段和方法以创建`Source`和`Properties`页面。

```
private TreeViewer treeViewer;
private TextEditor textEditor;

protected void createPages() {
    createPropertiesPage();
    createSourcePage();
    updateTitle();
}

void createPropertiesPage() {
    treeViewer = new TreeViewer(
        getContainer(), SWT.MULTI | SWT.FULL_SELECTION);
    int index = addPage(treeViewer.getControl());
    setPageText(index, "Properties");
}
```



```
void createSourcePage() {
    try {
        textEditor = new TextEditor();
        int index = addPage(textEditor, getEditorInput());
        setPageText(index, "Source");
    }
    catch (PartInitException e) {
        FavoritesLog.logError("Error creating nested text editor", e);
    }
}

void updateTitle() {
    IEditorInput input = getEditorInput();
    setPartName(input.getName());
    setTitleToolTip(input.getToolTipText());
}
```

当焦点移至编辑器时，将调用setFocus()方法。该方法必须随后根据当前选中的页面将焦点重定向至合适的编辑器。

```
public void setFocus() {
    switch (getActivePage()) {
        case 0:
            treeViewer.getTree().setFocus();
            break;
        case 1:
            textEditor.setFocus();
            break;
    }
}
```

当用户直接或间接请求显示一个标记时，要保证Source页面是活动的，然后将请求重定向至文本编辑器。当Properties页面是活动的时，你可以做一些不同的事。但是这将需要额外的编辑器模型结构。

```
public void gotoMarker(IMarker marker) {
    setActivePage(1);
    ((IGotoMarker) textEditor.getAdapter(IGotoMarker.class))
        .gotoMarker(marker);
}
```

这三个方法用于保存编辑器内容。如果isSaveAsAllowed()方法返回false，那么将不会调用doSaveAs()方法。

```
public boolean isSaveAsAllowed() {
    return true;
}

public void doSave(IProgressMonitor monitor) {
    textEditor.doSave(monitor);
}

public void doSaveAs() {
    textEditor.doSaveAs();
    setInput(textEditor.getEditorInput());
    updateTitle();
}
```

这些代码定义了一个十分简单的编辑器。当打开该编辑器时，第一个页面是一个空的树（将在下一节添加内容），而第二个页面是一个嵌入的文本编辑器（图8-3）。编辑器在第二个页面中使用

嵌入的文本编辑器处理所有的常规文本编辑操作。但第一个页面需要运行。

首先，你需要向树中添加列。这通过添加两个新字段和附加功能至createPropertiesPage()方法实现。为了使显示看起来更完美，与收藏夹视图被自动调整大小的方法类似，在树中自动调整列（参见7.8节）。

```
private TreeColumn keyColumn;
private TreeColumn valueColumn;

void createPropertiesPage() {
    Composite treeContainer = new Composite(getContainer(), SWT.NONE);
    TreeColumnLayout layout = new TreeColumnLayout();
    treeContainer.setLayout(layout);

    treeViewer = new TreeViewer(
        treeContainer, SWT.MULTI | SWT.FULL_SELECTION);
    Tree tree = treeViewer.getTree();
    tree.setHeaderVisible(true);

    keyColumn = new TreeColumn(tree, SWT.NONE);
    keyColumn.setText("Key");
    layout.setColumnData(keyColumn, new ColumnWeightData(2));

    valueColumn = new TreeColumn(tree, SWT.NONE);
    valueColumn.setText("Value");
    layout.setColumnData(valueColumn, new ColumnWeightData(3));

    int index = addPage(treeContainer);
    setPageText(index, "Properties");
}
```

当运行时，Properties编辑器将在Properties页面中显示两个空列（图8-4）。

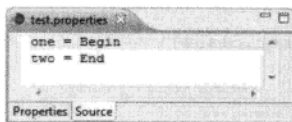


图8-3 属性编辑器的源代码页

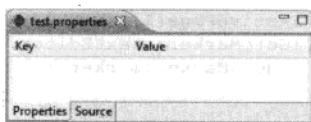


图8-4 属性编辑器的属性页

8.2.3 编辑器模型

下一步是完善树，以使文本编辑器中的内容出现于树中。为了完成这项任务，你需要创建一个可以分析文本编辑器内容的模型，然后将该模型和一个标签提供者一起，关联至树。当然，在该模型中有许多可以改进的地方，比如，展开分析结果，重构代码至一个单独的类，和增强分析程序以处理多行的值。然而，对于我们这里所要说明的问题来说，该模型已经足够了。

我们首先为所有属性模型对象引入一个新的PropertyElement超类。

```
public abstract class PropertyElement
{
    public static final PropertyElement[] NO_CHILDREN = {};
    private PropertyElement parent;

    public PropertyElement(PropertyElement parent) {
```

```
        this.parent = parent;
    }
    public PropertyElement getParent() {
        return parent;
    }
    public abstract PropertyElement[] getChildren();
    public abstract void removeFromParent();
}
```

一个PropertyEntry对象表示属性文件中的一个名/值对。请注意下面的三个类是相互依赖的，并且应被一起添加至你的项目。

```
public class PropertyEntry extends PropertyElement
{
    String key;
    String value;

    public PropertyEntry(
        PropertyCategory parent, String key, String value
    ) {
        super(parent);
        this.key = key;
        this.value = value;
    }
    public String getKey() {
        return key;
    }

    public String getValue() {
        return value;
    }

    public PropertyElement[] getChildren() {
        return NO_CHILDREN;
    }

    public void setKey(String text) {
        if (key.equals(text))
            return;
        key = text;
        ((PropertyCategory) getParent()).keyChanged(this);
    }

    public void setValue(String text) {
        if (value.equals(text))
            return;
        value = text;
        ((PropertyCategory) getParent()).valueChanged(this);
    }

    public void removeFromParent() {
        ((PropertyCategory) getParent()).removeEntry(this);
    }
}
```

一个PropertyCategory表示一个相关属性条目的组。在组前面的说明表示名称。该类别可以从一个读者对象（reader object）获取它的名称和条目。

```
package com.qualityclipse.favorites.editors;
import ...

public class PropertyCategory extends PropertyElement
{
    private String name;
    private List<PropertyEntry> .entries;

    public PropertyCategory(
        PropertyFile parent, LineNumberReader reader
    ) throws IOException {
        super(parent);

        // Determine the category name from comments.
        while (true) {
            reader.mark(1);
            int ch = reader.read();
            if (ch == -1)
                break;
            reader.reset();
            if (ch != '#')
                break;
            String line = reader.readLine();
            if (name == null) {
                line = line.replace('#', ' ').trim();
                if (line.length() > 0)
                    name = line;
            }
        }
        if (name == null)
            name = "";

        // Determine the properties in this category.
        entries = new ArrayList<PropertyEntry>();
        while (true) {
            reader.mark(1);
            int ch = reader.read();
            if (ch == -1)
                break;
            reader.reset();
            if (ch == '#')
                break;
            String line = reader.readLine();
            int index = line.indexOf('=');
            if (index != -1) {
                String key = line.substring(0, index).trim();
                String value = line.substring(index + 1).trim();
                entries.add(new PropertyEntry(this, key, value));
            }
        }
    }

    public String getName() {
        return name;
    }

    public Collection<PropertyEntry> getEntries() {
```

```
        return entries;
    }

    public PropertyElement[] getChildren() {
        return (PropertyElement[]) entries.toArray(
            new PropertyElement[entries.size()]);
    }

    public void setName(String text) {
        if (name.equals(text))
            return;
        name = text;
        ((PropertyFile) getParent()).nameChanged(this);
    }

    public void addEntry(PropertyEntry entry) {
        if (!entries.contains(entry)) {
            entries.add(entry);
            ((PropertyFile) getParent()).entryAdded(
                this, entry);
        }
    }

    public void removeEntry(PropertyEntry entry) {
        if (entries.remove(entry))
            ((PropertyFile) getParent()).entryRemoved(
                this, entry);
    }

    public void removeFromParent() {
        ((PropertyFile) getParent()).removeCategory(this);
    }

    public void keyChanged(PropertyEntry entry) {
        ((PropertyFile) getParent()).keyChanged(this, entry);
    }

    public void valueChanged(PropertyEntry entry) {
        ((PropertyFile) getParent()).valueChanged(this, entry);
    }
}
```

PropertyFile对象将这些联系为一个整体。

```
package com.qualityeclipse.favorites.editors;

import ...

import com.qualityeclipse.favorites.FavoritesLog;

public class PropertyFile extends PropertyElement
{
    private PropertyCategory unnamedCategory;
    private List<PropertyCategory> categories;
    private List<PropertyFileListener> listeners;

    public PropertyFile(String content) {
```




```
super(null);
categories = new ArrayList<PropertyCategory>();
listeners = new ArrayList<PropertyFileListener>();

LineNumberReader reader =
    new LineNumberReader(new StringReader(content));
try {
    unnamedCategory = new PropertyCategory(this, reader);
    while (true) {
        reader.mark(1);
        int ch = reader.read();
        if (ch == -1)
            break;
        reader.reset();
        categories.add(
            new PropertyCategory(this, reader));
    }
} catch (IOException e) {
    FavoritesLog.logError(e);
}
}

public PropertyElement[] getChildren() {
    List<PropertyElement> children
        = new ArrayList<PropertyElement>();
    children.addAll(unnamedCategory.getEntries());
    children.addAll(categories);
    return children.toArray(new PropertyElement[children.size()]);
}

public void addCategory(PropertyCategory category) {
    if (!categories.contains(category)) {
        categories.add(category);
        categoryAdded(category);
    }
}

public void removeCategory(PropertyCategory category) {
    if (categories.remove(category))
        categoryRemoved(category);
}

public void removeFromParent() {
    // Nothing to do.
}

void addPropertyFileListener(
    PropertyFileListener listener) {
    if (!listeners.contains(listener))
        listeners.add(listener);
}

void removePropertyFileListener(
    PropertyFileListener listener) {
    listeners.remove(listener);
}
```

```
}

void keyChanged(PropertyCategory category,PropertyEntry entry) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().keyChanged(category, entry);
}

void valueChanged(PropertyCategory category, PropertyEntry entry)
{
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().valueChanged(category, entry);
}

void nameChanged(PropertyCategory category) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().nameChanged(category);
}

void entryAdded(PropertyCategory category,PropertyEntry entry) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().entryAdded(category, entry);
}

void entryRemoved(PropertyCategory category, PropertyEntry entry)
{
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().entryRemoved(category, entry);
}

void categoryAdded(PropertyCategory category) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().categoryAdded(category);
}

void categoryRemoved(PropertyCategory category) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().categoryRemoved(category);
}
}
```

PropertyFile使用PropertyFileListener接口通知注册的监听器(如PropertiesEditor)模型中已经发生了更改。

```
package com.qualityeclipse.favorites.editors;

public interface PropertyFileListener
{
    void keyChanged(
        PropertyCategory category,
        PropertyEntry entry);
}
```

```

void valueChanged(
    PropertyCategory category,
    PropertyEntry entry);

void nameChanged(
    PropertyCategory category);

void entryAdded(
    PropertyCategory category,
    PropertyEntry entry);

void entryRemoved(
    PropertyCategory category,
    PropertyEntry entry);

void categoryAdded(
    PropertyCategory category);

void categoryRemoved(
    PropertyCategory category);
}

```

8.2.4 内容提供者

所有这些模型对象，仅当它们可以在树中正确显示时才有用。为了完成这项任务，你需要创建一个内容提供者和标签提供者。内容提供者提供树中出现的行和父/子关系，但不提供实际的单元内容。

```

package com.qualityclipse.favorites.editors;

import ...

public class PropertiesEditorContentProvider
    implements ITreeContentProvider
{
    public void inputChanged(
        Viewer viewer, Object oldInput, Object newInput
    ) { }

    public Object[] getElements(Object element) {
        return getChildren(element);
    }

    public Object[] getChildren(Object element) {
        if (element instanceof PropertyElement)
            return ((PropertyElement) element).getChildren();
        return null;
    }

    public Object getParent(Object element) {
        if (element instanceof PropertyElement)
            return ((PropertyElement) element).getParent();
        return null;
    }

    public boolean hasChildren(Object element) {
        if (element instanceof PropertyElement)
            return ((PropertyElement) element).getChildren().length > 0;
        return false;
    }
}

```

```
    }  
  
    public void dispose() {  
    }  
}
```

8.2.5 标签提供者

标签提供者转换由内容提供者返回的行元素对象为可以在表的单元格中显示的图像和文本。

```
package com.qualityeclipse.favorites.editors;  
  
import ...  
  
public class PropertiesEditorLabelProvider extends LabelProvider  
    implements ITableLabelProvider  
{  
    public Image getColumnImage(Object element, int columnIndex) {  
        return null;  
    }  
  
    public String getColumnText(Object element, int columnIndex) {  
        if (element instanceof PropertyCategory) {  
            PropertyCategory category =  
                (PropertyCategory) element;  
            switch (columnIndex) {  
                case 0 :  
                    return category.getName();  
                case 1 :  
                    return "";  
            }  
        }  
  
        if (element instanceof PropertyEntry) {  
            PropertyEntry entry = (PropertyEntry) element;  
            switch (columnIndex) {  
                case 0 :  
                    return entry.getKey();  
                case 1 :  
                    return entry.getValue();  
            }  
        }  
  
        if (element == null)  
            return "<null>";  
        return element.toString();  
    }  
}
```

最后，你需要添加一个新的initTreeContent()方法（在createPages()方法中调用），以将新的内容和标签提供者关联至树。该方法后面还跟随了一个新方法。该新方法用于将文本编辑器的内容和树的内容同步。对asyncExec()的调用保证updateTreeFromTextEditor方法是在UI线程中执行（参见4.2.5节以了解更多关于UI线程的内容）。updateTreeFromTextEditor()方法间接引用了org.eclipse.jface.text插件中的代码，所以它必须被添加至Favorites插件清单（图2-10）。

```
private PropertiesEditorContentProvider treeContentProvider;  
private PropertiesEditorLabelProvider treeLabelProvider;
```

```

void initTreeContent() {
    treeContentProvider = new PropertiesEditorContentProvider();
    treeViewer.setContentProvider(treeContentProvider);
    treeLabelProvider = new PropertiesEditorLabelProvider();
    treeViewer.setLabelProvider(treeLabelProvider);

    // Reset the input from the text editor's content
    // after the editor initialization has completed.
    treeViewer.setInput(new PropertyFile(""));
    treeViewer.getTree().getDisplay().asyncExec(new Runnable() {
        public void run() {
            updateTreeFromTextEditor();
        }
    });
    treeViewer.setAutoExpandLevel(TreeViewer.ALL_LEVELS);
}

void updateTreeFromTextEditor() {
    PropertyFile propertyFile = new PropertyFile(
        textEditor
            .getDocumentProvider()
            .getDocument(textEditor.getEditorInput())
            .get());
    treeViewer.setInput(propertyFile);
}

```

当所有这些任务都已经完成后，Properties编辑器的Properties页将包含一些内容（图8-5）。

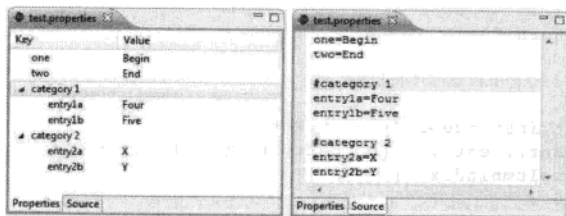


图8-5 包含新内容的属性编辑器

8.3 编辑

当Properties页在树中显示内容时，不需要切换至源代码页就可以编辑内容是十分重要的（参见14.2.4节以了解关于在已有文本编辑器中处理内容的示例）。

8.3.1 单元格编辑器

与创建createInlineEditor方法类似（参见7.3.9节），创建一个新的initTreeEditors()方法。createPages()将调用它。该方法初始化两个TreeViewerColumn实例用于在名称和值两个列中分别管理单元格编辑器：

```

TreeViewerColumn column1 =
    new TreeViewerColumn(treeViewer, keyColumn);
TreeViewerColumn column2 =
    new TreeViewerColumn(treeViewer, valueColumn);

```

每一个TreeViewerColumn具有一个与其关联的ColumnLabelProvider:

```
column1.setLabelProvider(new ColumnLabelProvider() {
    public String getText(Object element) {
        return treeLabelProvider.getColumnText(element, 0);
    }
});
column2.setLabelProvider(new ColumnLabelProvider() {
    public String getText(Object element) {
        return treeLabelProvider.getColumnText(element, 1);
    }
});
```

在第一列中, 用户可以编辑类别的名称或名/值对的名称。EditingSupport用于初始化一个合适的单元格编辑器, 为单元格编辑器获取合适的文本, 并保存修改过的文本至模型。

```
column1.setEditingSupport(new EditingSupport(treeViewer) {
    TextCellEditor editor = null;

    protected boolean canEdit(Object element) {
        return true;
    }

    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
        }
        return editor;
    }

    protected Object getValue(Object element) {
        return treeLabelProvider.getColumnText(element, 0);
    }

    protected void setValue(Object element, Object value) {
        String text = ((String) value).trim();
        if (element instanceof PropertyCategory)
            ((PropertyCategory) element).setName(text);
        if (element instanceof PropertyEntry)
            ((PropertyEntry) element).setKey(text);
    }
});
```

我们为第二列创建一个类似的EditingSupport对象, 但修改它以允许编辑名/值对中的值:

```
column2.setEditingSupport(new EditingSupport(treeViewer) {
    TextCellEditor editor = null;

    protected boolean canEdit(Object element) {
        return element instanceof PropertyEntry;
    }

    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
        }
    }
});
```

```

        return editor;
    }

    protected Object getValue(Object element) {
        return treeLabelProvider.getColumnText(element, 1);
    }

    protected void setValue(Object element, Object value) {
        String text = ((String) value).trim();
        if (element instanceof PropertyEntry)
            ((PropertyEntry) element).setValue(text);
    }
}

```

setValue方法改变编辑器模型(图8-6), 而该模型稍后调用PropertiesEditor类的新的treeModified()方法, 以通知所有感兴趣的成员: “编辑器的内容已经被修改了”。这项功能通过在下一节创建的新的PropertyFile Listener监听器实现。

```

    public void treeModified() {
        if (!isDirty())
            firePropertyChange(IEditorPart.PROP_DIRTY);
    }

```

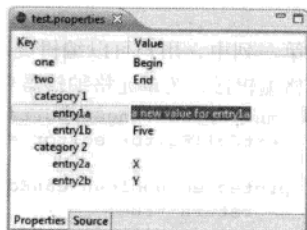


图8-6 具有修改过的单元格的属性编辑器

8.3.2 变更监听器

当用户编辑值时, 模型生成一个变更事件以通知注册的监听器。下一步是将一个变更监听器关联于PropertiesEditor类中以使得可以将事件通知你, 并恰当地更新树。首先, 添加一个新的PropertyFile Listener。

```

private final PropertyFileListener propertyFileListener =
    new PropertyFileListener()
{
    public void keyChanged(
        PropertyCategory category, PropertyEntry entry
    ) {
        treeViewer.refresh(entry);
        treeModified();
    }

    public void valueChanged(
        PropertyCategory category, PropertyEntry entry
    ) {
        treeViewer.refresh(entry);
        treeModified();
    }

    public void nameChanged(PropertyCategory category) {
        treeViewer.refresh(category);
        treeModified();
    }

    public void entryAdded(
        PropertyCategory category, PropertyEntry entry
    ) {

```



```

    ) {
        treeViewer.refresh();
        treeModified();
    }

    public void entryRemoved(
        PropertyCategory category, PropertyEntry entry
    ) {
        treeViewer.refresh();
        treeModified();
    }

    public void categoryAdded(PropertyCategory category) {
        treeViewer.refresh();
        treeModified();
    }

    public void categoryRemoved(PropertyCategory category) {
        treeViewer.refresh();
        treeModified();
    }
};

```

然后, 根据以下内容修改updateTreeFromTextEditor()方法, 使得监听器在它被丢弃前从旧的编辑器中移除并添加至新的编辑器模型。

```

void updateTreeFromTextEditor() {
    PropertyFile propertyFile =(PropertyFile)treeViewer.getInput();
    propertyFile.removePropertyFileListener(propertyFileListener);
    propertyFile = new PropertyFile(
        textEditor
            .getDocumentProvider()
            .getDocument(textEditor.getEditorInput())
            .get());
    treeViewer.setInput(propertyFile);
    propertyFile.addPropertyFileListener(propertyFileListener);
}

```

8.3.3 单元格验证器

单元格编辑器具有验证器以阻止不合法的输入访问模型对象。无论何时用户修改单元格编辑器的内容, isValid(Object)方法当对象代表一个非法值时返回一个错误消息, 当值是合法时返回null。根据下列内容在initTreeEditors()方法中给列中的单元格编辑器分配验证器。

```

column1.setEditingSupport(new EditingSupport(treeViewer) {
    ...
    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
            editor.setValidator(new ICellEditorValidator(){
                public String isValid(Object value){
                    if (((String)value).trim().length()==0)
                        return "Key must not be empty string";
                    return null;
                }
            });
        }
    }
});

```



```

    }
    return editor;
}

```

无论何时单元格验证器返回一个错误消息，都将使用null值调用setValue方法。添加代码至setValue方法以防止出现该情况。

```

column1.setEditingSupport(new EditingSupport(treeViewer) {
    ...
    protected void setValue(Object element, Object value) {
        if (value == null)
            return;
        String text = ((String) value).trim();
        if (element instanceof PropertyCategory)
            ((PropertyCategory) element).setName(text);
        if (element instanceof PropertyEntry)
            ((PropertyEntry) element).setKey(text);
    }
    ...
}

```

无论何时用户输入一个不合法的值，你必须决定如何通知该用户该值是不合法的。在这种情况下，在initTreeEditors()方法中添加一个ICellEditorListener以使在窗口状态栏中显示错误信息(图8-7)。要获得一个更突出的错误消息，可以重新设计编辑器的头部区域以允许在树的上方显示一个错误图像和消息，而不是在工作台的状态栏。

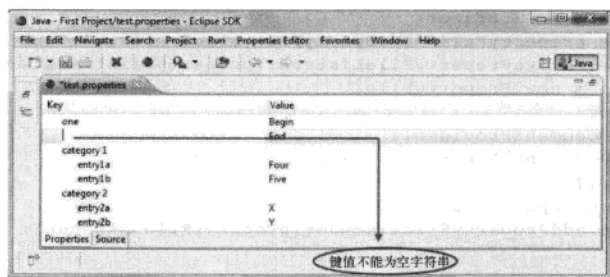


图8-7 表示不合法的输入的状态栏错误消息

```

column1.setEditingSupport(new EditingSupport(treeViewer) {
    ...
    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
            ...
            editor.addListener(new ICellEditorListener(){
                public void applyEditorValue(){
                    setErrorMessage(null);
                }
                public void cancelEditor(){
                    setErrorMessage(null);
                }
                public void editorValueChanged(
                    boolean oldValidState, boolean newValidState){

```

```
        setErrorMessage(editor.getErrorMessage());
    }
    private void setErrorMessage(String errorMessage){
        getEditorSite().getActionBars()
            .getStatusLineManager()
            .setErrorMessage(errorMessage);
    }
    });
    ...
```

8.3.4 编辑与选择

在编辑被添加至树之前，用户可以很容易的选择一个或多个行。但现在单元格编辑器是一直打开的。一个可能的解决办法是仅当Alt键被按下时才打开编辑器，但选择一行或多行则不打开编辑器。为了实现该功能，从收藏夹视图中释放ColumnViewerEditorActivationListener（参见7.3.9节的结尾部分）至一个新的AltClickCellEditListener类。然后添加以下内容至initTreeEditors()方法的结尾部分。

```
treeViewer.getColumnViewerEditor().addEditorActivationListener(
    new AltClickCellEditListener());
```

8.4 编辑器生命周期

一般的编辑器具有打开—修改—保存—关闭的生命周期。当打开编辑器时，将调用init(IEditorSite, IEditorInput)方法以设置编辑器的初始内容。当用户修改该编辑器的内容时，编辑器通过使用firePropertyChange(int)方法必须通知其他部分，它的内容现在是“脏的”。当一个用户保存编辑器的内容时，必须再次使用firePropertyChange(int)方法以通知注册的监听器，编辑器的内容不再是脏的了。Eclipse根据由isDirty()返回的值自动注册监听器至不同的平台任务，如更新编辑器的标题，在标签前添加或移除一个*号，以及使保存菜单可用等。最后，当编辑器被关闭时，如果isDirty()方法返回true，编辑器的内容将被保存。

8.4.1 修改过的编辑器

你需要确保编辑器知道它的内容自从上次保存操作后是否被用户修改过。要实现该功能，引入该新字段以跟踪当前页是否相对于其他页面已经被修改过（dirty）：

```
private boolean isPageModified;
```

无论何时修改当前页面的内容，你都需要设置新的isPageModified字段。无论何时树被修改了，单元格编辑器都将调用treeModified()方法（参见8.3.1节）。在该方法中可以设置新的isPageModified字段。

```
public void treeModified() {
    boolean wasDirty = isDirty();
    isPageModified = true;
    if (!wasDirty)
        firePropertyChange(IEditorPart.PROP_DIRTY);
}
```

无论何时修改了文本编辑器，MultiPageEditorPart的addPage()方法都将使用handlePropertyChange(int)方法（参见8.2.2节中的createSourcePage()方法）以当编辑器内容改变时通知其他部分。你可以恰当地覆盖该方法以设置isPageModified字段：

```
protected void handlePropertyChange (int propertyId) {
    if (propertyId == IEditorPart.PROP_DIRTY)
```

```

        isPageModified = isDirty();
        super.handlePropertyChange(propertyId);
    }

```

最后，你需要在编辑器内容是修改过的时候让其他已注册的监听器知道。MultiPageEditorPart的isDirty()方法恰当地为Source页面的嵌套文本编辑器返回true，但它完全不知道对树做出的更改。覆盖该方法以添加这项内容将使得Save菜单项被设置为可用，并且在合适的时间更新编辑器的标题。

```

    public boolean isDirty() {
        return isPageModified || super.isDirty();
    }

```

8.4.2 切换页面

当在Properties和Source页面切换时，所有在Properties页面中做出的编辑必须自动同步至Source页面，反之亦然。要实现该功能，根据以下内容覆盖pageChange(int)方法以更新页面内容：

```

protected void pageChange(int newPageIndex) {
    switch (newPageIndex) {
        case 0 :
            if (isDirty())
                updateTreeFromTextEditor();
            break;
        case 1 :
            if (isPageModified)
                updateTextEditorFromTree();
            break;
    }
    isPageModified = false;
    super.pageChange(newPageIndex);
}

```

我们已经定义了updateTreeFromTextEditor()方法（参见8.2.3节），但updateTextEditorFromTree()方法还没有，所以我们现在添加它。

```

void updateTextEditorFromTree() {
    textEditor
        .getDocumentProvider()
        .getDocument(textEditor.getEditorInput())
        .set(((PropertyFile) treeViewer.getInput()).asText());
}

```

updateTextEditorFromTree()方法调用PropertyFile中的新的asText()方法。该asText()方法通过重新组合模型为一个文本表示反转PropertyFile的构造函数中的解析过程（参见8.2.3节）。

```

public String asText() {
    StringWriter stringWriter = new StringWriter(2000);
    PrintWriter writer = new PrintWriter(stringWriter);
    unnamedCategory.appendText(writer);
    Iterator<PropertyCategory> iter = categories.iterator();
    while (iter.hasNext()) {
        writer.println();
        iter.next().appendText(writer);
    }
    return stringWriter.toString();
}

```

asText()方法调用PropertyCategory中的新的appendText(PrintWriter)方法：

```
public void appendText(PrintWriter writer) {
    if (name.length() > 0) {
        writer.print("# ");
        writer.println(name);
    }
    Iterator<PropertyEntry> iter = entries.iterator();
    while (iter.hasNext())
        iter.next().appendText(writer);
}
```

该方法然后将调用PropertyEntry中新的appendText(PrintWriter)方法:

```
public void appendText(PrintWriter writer) {
    writer.print(key);
    writer.print(" = ");
    writer.println(value);
}
```

8.4.3 保存内容

由于当前实现使用了嵌套的文本编辑器来保存内容至当前被编辑的文件, Properties页面中的更改将不会被发觉除非用户切换至Source页面。必须修改以下方法以在保存前更新嵌套的文本编辑器。由于保存操作一般都是长时间的操作, 进度监视器用于与用户交流进度 (参见9.4节)。

```
public void doSave(IProgressMonitor monitor) {
    if (getActivePage() == 0 && isPageModified)
        updateTextEditorFromTree();
    isPageModified = false;
    textEditor.doSave(monitor);
}

public void doSaveAs() {
    if (getActivePage() == 0 && isPageModified)
        updateTextEditorFromTree();
    isPageModified = false;
    textEditor.doSaveAs();
    setInput(textEditor.getEditorInput());
    updateTitle();
}
```

8.5 编辑器命令

编辑器命令可以作为菜单项出现于编辑器的上下文菜单中, 可以作为工具栏按钮出现于工作台工具栏中, 也可以作为菜单项出现于工作台菜单中 (图6-17)。本节包含了在程序中添加操作至编辑器, 而6.9节讨论了通过在插件清单中使用声明添加操作 (参见14.2.4节以了解关于操作已有的文本编辑器的内容的示例)。

8.5.1 上下文菜单

一般地, 编辑器具有由目标为编辑器或编辑器中选对象的命令生成的上下文菜单。创建编辑器的上下文菜单包含几个步骤, 还需要几个额外步骤以注册编辑器, 以使他人可以添加命令 (参见6.2.5节和6.2.7节以了解关于命令是如何通过插件清单被添加项至编辑器的上下文菜单的内容)。

1. 创建上下文菜单

上下文菜单必须和编辑器同时创建。然而, 由于添加者可以根据选择添加或移除菜单项, 它的

内容直到在用户点击鼠标右键之后，在菜单显示之前才能被确定。为了实现该功能，设置菜单的 `RemoveAllWhenShown` 属性为 `true` 以使菜单每次都重新创建，并添加一个菜单监听器以动态地创建菜单。此外，菜单必须被注册至控件，以使它可以被显示。菜单还必须被注册至编辑器站点，以使其他插件可以向其添加项命令（参见6.2节）。

对于 `Properties` 编辑器而言，修改 `createPages()` 以调用该 `createContextMenu()` 方法。该方法的最后一行语句注册了菜单以使其他插件可以向其添加项。

```
private void createContextMenu() {
    MenuManager menuMgr = new MenuManager("#PopupMenu");
    menuMgr.setRemoveAllWhenShown(true);
    menuMgr.addMenuListener(new IMenuListener() {
        public void menuAboutToShow(IMenuManager m) {
            PropertiesEditor.this.fillContextMenu(m);
        }
    });
    Tree tree = treeViewer.getTree();
    Menu menu = menuMgr.createContextMenu(tree);
    tree.setMenu(menu);
    getSite().registerContextMenu(menuMgr, treeViewer);
}
```

2. 动态创建上下文菜单

每当用户点击鼠标右键时，上下文菜单的内容必须被重新创建。这是因为添加者可以根据编辑器的选择添加命令。此外，上下文菜单必须包含一个具有“edit”的分割线用于我们自己的命令和另一个具有 `IWorkbenchActionConstants.MB_ADDITIONS` 的分割线，表示那些被添加项的操作将会出现于上下文菜单中。`createContextMenu()` 方法（参见上一节）调用该新的 `fillContextMenu(IMenuManager)` 方法：

```
private void fillContextMenu(IMenuManager menuMgr) {
    menuMgr.add(new Separator("edit"));
    menuMgr.add(
        new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
}
```

3. 创建命令

`Properties` 编辑器需要一个命令用于从编辑器删除选中的树元素。我们可以创建一个添加项（参见7.3.2节）。但是，在插件清单中声明 `command` 和 `menuContribution` 要简单得多。我们从声明一个具有以下属性的 `command`（参见6.1.1节）开始：

- `id`——“com.qualityeclipse.properties.editor.delete”
- `name`——“Delete”
- `description`——“Delete the selected entries in the Properties Editor”
- `categoryId`——“com.qualityeclipse.favorites.commands.category”
- `defaultHandler`——“com.qualityeclipse.favorites.handlers.DeletePropertiesHandler”

为了创建上面引用的处理器，点击该属性的文本字段左侧的 `defaultHandler:` 标签以打开 `New Java Class` 向导（为了了解更多信息，参见6.3节）。

```
public class DeletePropertiesHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        ISelection selection = HandlerUtil.getCurrentSelection(event);
```

```
if (!(selection instanceof IStructuredSelection))
    return null;
Iterator iter = ((IStructuredSelection) selection).iterator();
Shell shell = HandlerUtil.getActiveShellChecked(event);
shell.setRedraw(false);
try {
    while (iter.hasNext())
        ((PropertyElement) iter.next()).removeFromParent();
}
finally {
    shell.setRedraw(true);
}
return null;
}
```

提示 根据上面代码所显示的, 使用shell的setRedraw(boolean)方法以在对一个控件或它的模型做出多个更改时降低闪烁。

上面描述的DeletePropertiesHandler要获取它的选择, Properties编辑器的树查看器必须是一个选择提供者(参见7.4节)。要实现该功能, 添加以下语句至createPropertiesPage方法的末尾:

```
getSite().setSelectionProvider(treeViewer);
```

为了在Properties编辑器的上下文菜单中显示新的命令, 添加一个具有以下locationURI的menuContribution(参见6.2.7节):

```
popup:com.qualityeclipse.properties.editor?after=edit
```

右键点击新的menuContribution并选择New > command。根据下面所示设置新命令的属性, 然后添加一个visibleWhen表达式以防止当没有选中任何项时显示命令(参见6.2.10节)。

- commandId——“com.qualityeclipse.properties.editor.delete”
- icon——“icons/delete_edit.gif”

当实现了该功能后, 包含Delete菜单项和其他人的添加项的上下文菜单将会显示(图8-8)。

8.5.2 编辑器添加程序

编辑器添加程序管理一个或多个编辑器的全局菜单、菜单项和工具栏按钮的安装和移除。通过使用命令大幅度降低甚至消除了对于编辑器添加程序的需要。使用命令是推荐方法, 但基于内容完整的考虑, 我们包含这一节。使用命令取代编辑器添加程序提供同样的功能将在下一节中讨论(参见8.5.3节)。

org.eclipse.ui.IEditorActionBarContributor的实例与一个或多个编辑器, 添加或移除最高级菜单和工具栏元素相关联。插件清单指定了哪一个编辑器添加程序, 一般是org.eclipse.ui.part.EditorActionBarContributor或org.eclipse.ui.part.MultiPageEditorActionBarContributor的子类, 与哪一个编辑器类型关联(参见8.1节)。平台然后发送以下事件至添加程序, 表示当编辑器变成活动或不活动, 以使添加程序可以恰当地添加或移除菜单和按钮。

- dispose()——当不再需要添加程序时自动调用该方法。它清理所有的平台资源, 如图像、剪贴板等由该类创建的资源。该方法遵循谁创建就由谁销毁的Eclipse原则。

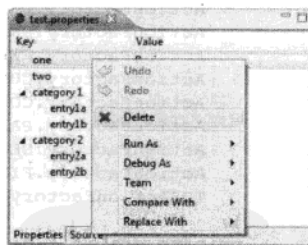


图8-8 属性编辑器的上下文菜单

- `init(IActionBars, IWorkbenchPage)`——当添加程序第一次被创建时调用该方法。
- `setActiveEditor(IEditorPart)`——当一个相关编辑器变成活动或不活动时调用该方法。添加程序应恰当的插入或删除菜单和工具栏按钮。
- `EditorActionBarContributor`类实现了 `IEditorActionBarContributor` 接口, 缓存操作栏和工作台页, 并提供两个新的访问方法。
- `getActionBars()`——返回当添加程序被初始化时提供给它的操作栏。
- `getPage()`——返回当添加程序被初始化时提供给它的工作台页面。

`MultiPageEditorActionBarContributor`方法扩展了 `EditorActionBarContributor`, 提供了一个取代 `setActiveEditor(IEditorPart)`方法的新方法以用于覆盖。

- `setActivePage(IEditorPart)`——设置多页面编辑器的活动页面为给定编辑器。如果没有活动页面, 或活动页面不具有一个对应的编辑器, 参数为 `null`。

1. 全局操作

通过从 `org.eclipse.ui.editors.text.TextEditorActionContributor` 和 `org.eclipse.ui.texteditor.BasicTextEditorActionContributor` 借鉴, 你将创建你自己的用于 `Properties` 编辑器的添加程序。该添加程序不仅恰当将全局操作 (如 `cut`, `copy`, `paste` 等位于 `Edit` 菜单中的命令) 关联至活动编辑器, 还关联至编辑器中的活动页面。

```
package com.qualityeclipse.favorites.editors;

import ...

public class PropertiesEditorContributor
    extends EditorActionBarContributor
{
    private static final String[] WORKBENCH_ACTION_IDS = {
        ActionFactory.DELETE.getId(),
        ActionFactory.UNDO.getId(),
        ActionFactory.REDO.getId(),
        ActionFactory.CUT.getId(),
        ActionFactory.COPY.getId(),
        ActionFactory.PASTE.getId(),
        ActionFactory.SELECT_ALL.getId(),
        ActionFactory.FIND.getId(),
        IDEActionFactory.BOOKMARK.getId(),
    };
    private static final String[] TEXTEDITOR_ACTION_IDS = {
        ActionFactory.DELETE.getId(),
        ActionFactory.UNDO.getId(),
        ActionFactory.REDO.getId(),
        ActionFactory.CUT.getId(),
        ActionFactory.COPY.getId(),
        ActionFactory.PASTE.getId(),
        ActionFactory.SELECT_ALL.getId(),
        ActionFactory.FIND.getId(),
        IDEActionFactory.BOOKMARK.getId(),
    };

    public void setActiveEditor(IEditorPart part) {
        PropertiesEditor editor = (PropertiesEditor) part;
```

```

        setActivePage(editor, editor.getActivePage());
    }
    public void setActivePage(
        PropertiesEditor editor,
        int pageIndex
    ) {
        IActionBars actionBars = getActionBars();
        if (actionBars != null) {
            switch (pageIndex) {
                case 0 :
                    hookGlobalTreeActions(editor, actionBars);
                    break;
                case 1 :
                    hookGlobalTextActions(editor, actionBars);
                    break;
            }
            actionBars.updateActionBars();
        }
    }
    private void hookGlobalTreeActions(
        PropertiesEditor editor,
        IActionBars actionBars
    ) {
        for (int i = 0; i < WORKBENCH_ACTION_IDS.length; i++)
            actionBars.setGlobalActionHandler(
                WORKBENCH_ACTION_IDS[i],
                editor.getTreeAction(WORKBENCH_ACTION_IDS[i]));
    }

    private void hookGlobalTextActions(
        PropertiesEditor editor,
        IActionBars actionBars
    ) {
        ITextEditor textEditor = editor.getSourceEditor();
        for (int i = 0; i < WORKBENCH_ACTION_IDS.length; i++)
            actionBars.setGlobalActionHandler(
                WORKBENCH_ACTION_IDS[i],
                textEditor.getAction(TEXTEDITOR_ACTION_IDS[i]));
    }
}

```

现在，修改Properties编辑器以向添加程序增加访问方法。

```

public ITextEditor getSourceEditor() {
    return textEditor;
}

public IAction getTreeAction(String workbenchActionId) {
    if (ActionFactory.DELETE.getId().equals(workbenchActionId))
        return removeAction;
    return null;
}

```

添加以下行至pageChange()方法的末尾以当页面更改时通知添加程序，以使它可以恰当地更新菜单项和工具栏按钮。

```

IEditorActionBarContributor contributor =
    getEditorSite().getActionBarContributor();

```



```

if (contributor instanceof PropertiesEditorContributor)
    ((PropertiesEditorContributor) contributor)
        .setActivePage(this, newPageIndex);

```

2. 最高级菜单

然后，添加Delete操作至最高级菜单以显示它是如何被完成的。在这种情况下，与上下文菜单那样直接引用操作（参见8.5.1节）不同，你将使用org.eclipse.ui.actions.RetargetAction的实例，或更明确的org.eclipse.ui.actions.LabelRetargetAction。它将通过它的标识符间接引用移除操作。你将会使用ActionFactory.DELETE.getId()标识符，但只要setGlobalActionHandler(String,IAction)被用于将标识符与操作关联，那么就可以使用任意标识符。要实现所有这些功能，添加以下内容至PropertiesEditorContributor。

```

private LabelRetargetAction retargetRemoveAction =
    new LabelRetargetAction(ActionFactory.DELETE.getId(), "Remove");

public void init(IActionBars bars, IWorkbenchPage page) {
    super.init(bars, page);
    page.addPartListener(retargetRemoveAction);
}

public void contributeToMenu(IMenuManager menuManager) {
    IMenuManager menu = new MenuManager("Property Editor");
    menuManager.prependToGroup(
        IWorkbenchActionConstants.MB_ADDITIONS,
        menu);
    menu.add(retargetRemoveAction);
}

public void dispose() {
    getPage().removePartListener(retargetRemoveAction);
    super.dispose();
}

```

3. 工具栏按钮

你可以使用同样的被重定向的操作（参见上一节），并通过在PropertiesEditorContributor中包含以下代码，添加按钮至工作台的工具栏。

```

public void contributeToToolBar(IToolBarManager manager) {
    manager.add(new Separator());
    manager.add(retargetRemoveAction);
}

```

4. 键盘操作

通过再一次使用remove操作（参见8.5.1节），你可以通过修改稍早介绍的initTreeEditors()方法关联Delete键，这样当用户按下它时，在树中被选中的属性名/值对将会被移除。

```

private void initTreeEditors() {
    ... existing code ...
    treeViewer.getTree().addKeyListener(new KeyListener() {
        public void keyPressed(KeyEvent e) {
            if (e.keyCode == SWT.ALT)
                isAltPressed = true;
            if (e.character == SWT.DEL)
                removeAction.run();
        }
        public void keyReleased(KeyEvent e) {

```

```
        if (e.keyCode == SWT.ALT)
            isAltPressed = false;
    }
    });
}
```

8.5.3 编辑器命令而不是编辑器添加程序

命令API相对于操作是更好的选择，并且它降低或消除了对于编辑器添加程序的要求。你可以添加最高级菜单项（参见6.2.1节）或最高级工具栏（参见6.2.3节），并限制它的可见性（参见6.2.4节）。这样，仅当你的编辑器是活动的时，它才出现。在这种情况下，我们重新实现了上一节同样的功能（参见8.5.2节），除了使用命令之外。

1. 全局命令

在此时，全局命令（比如，Edit菜单中的cut、copy、paste等）在我们的Properties编辑器的“Source”页面能正常工作，但在“Properties”页面不能。要为Edit > Delete命令添加支持，使用以下插件声明将DeletePropertiesHandler（参见8.5.1节）与全局菜单项关联（参见6.3节以了解更多信息）。

```
<handler
    class="com.qualityeclipse.favorites.handlers.DeletePropertiesHandler"
    commandId="org.eclipse.ui.edit.delete">
    <activeWhen>
        <with
            variable="activeEditorId">
            <equals
                value="com.qualityeclipse.properties.editor">
            </equals>
        </with>
    </activeWhen>
</handler>
```

2. 最高级菜单

接下来，添加Delete操作至最高级菜单。这是出于显示它是如何被完成的目的。为了实现该功能，添加一个删除菜单项（参见6.2.2节）和一个visibleWhen表达式（参见6.2.10节）。

```
<menuContribution
    locationURI="menu:org.eclipse.ui.main.menu?after=additions">
    <menu
        id="com.qualityeclipse.favorites.menus.PropertiesEditor"
        label="Properties Editor">
        <command
            commandId="com.qualityeclipse.properties.editor.delete"
            icon="icons/delete_edit.gif">
        </command>
        <visibleWhen
            checkEnabled="false">
            <with
                variable="activeEditorId">
                <equals
                    value="com.qualityeclipse.properties.editor">
                </equals>
            </with>
        </visibleWhen>
    </menu>
</menuContribution>
```

以上可以正常工作。但是当在属性视图中即使没有东西被选中时，Delete菜单项也能使用。向处理器添加一个enabledWhen表达式将能起作用，但处理器当前是命令的默认处理器，并且enabledWhen表达式不能被关联至命令。移动处理器至一个单独的声明并关联一个enableWhen表达式（参见6.3节）。

```
<handler
  class="com.qualityclipse.favorites.handlers.DeletePropertiesHandler"
  commandId="com.qualityclipse.properties.editor.delete">
  <enabledWhen>
    <with
      variable="selection">
        <count
          value="+ ">
        </count>
      </with>
    </enabledWhen>
  </handler>
```

当完成以上内容后，这些代码将生成一个新的最高级菜单以出现于工作台的菜单栏中（图8-9）。

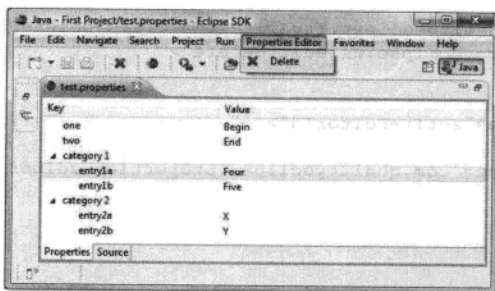


图8-9 属性编辑器菜单

3. 工具栏按钮

你可以使用与上一节列出的类似的方法添加最高级工具栏按钮（参见6.2.3节）。visibleWhen表达式保证工具栏添加程序仅当编辑器是活动时才可见。

```
<menuContribution
  locationURI="toolbar:org.eclipse.ui.main.toolbar?after=additions">
  <toolbar
    id="com.qualityclipse.properties.editor.toolbar">
    <command
      commandId="com.qualityclipse.properties.editor.delete"
      icon="icons/delete_edit.gif">
    </command>
    <visibleWhen
      checkEnabled="false">
      <with
        variable="activeEditorId">
        <equals
          value="com.qualityclipse.properties.editor">
        </equals>
      </with>
```

```

        </visibleWhen>
    </toolbar>
</menuContribution>

```

4. 键盘命令

编辑器键盘绑定上下文降低了键盘快捷键的复杂程度（参见6.4节）。这些快捷键仅当特定编辑器是活动的时才可用。为了展示这项技术，我们的目标是将F9键与删除命令关联起来。这样，当Properties编辑器具有焦点时，按下F9键将删除当前被选中的属性文件条目。我们从创建一个新的Properties编辑器键绑定上下文开始。

```

<extension point="org.eclipse.ui.contexts">
    <context
        id="com.qualityeclipse.properties.editor.context"
        name="Properties Editor Context"
        parentId="org.eclipse.ui.textEditorScope"/>
</extension>

```

该上下文应仅当Properties编辑器拥有焦点时才是活动的。为了实现该功能，添加一个从createPages方法调用的新方法以当Properties编辑器的树获得和失去焦点时，在程序中分别设置新的键绑定上下文为可用和不可用。

```

private void initKeyBindingContext() {
    final IContextService service = (IContextService)
        getSite().getService(IContextService.class);

    treeViewer.getControl().addFocusListener(new FocusListener() {
        IContextActivation currentContext = null;

        public void focusGained(FocusEvent e) {
            if (currentContext == null)
                currentContext = service.activateContext(
                    "com.qualityeclipse.properties.editor.context ");
        }

        public void focusLost(FocusEvent e) {
            if (currentContext != null)
                service.deactivateContext(currentContext);
        }
    });
}

```

最后，将F9键独占地与删除命令关联。

```

<extension
    point="org.eclipse.ui.bindings">
    <key
        commandId="com.qualityeclipse.properties.editor.delete"
        contextId="com.qualityeclipse.properties.editor.context"
        schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"
        sequence="F9">
    </key>
</extension>

```

8.5.4 撤销/重做

为用户添加撤销和重做命令的功能包括分离用户编辑为在用户界面上可见的命令和可以被执

行、撤销和重做的底层操作。一般地，每一个命令每当用户触发它时都将初始化一个操作。命令收集当前的程序状态，如当前被选中的元素和缓存该状态的操作使得它可以独立于原始命令而被执行、撤销和重做。IOperationHistory的实例管理全局撤销/重做栈中的操作（图8-10）。每一个操作使用一个或多个关联的撤销/重做上下文用于不同部分独立保存操作。

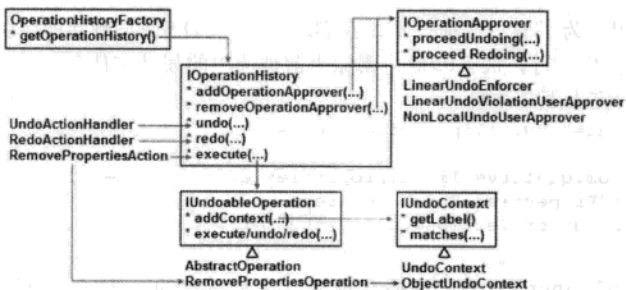


图8-10 Eclipse撤销/重做基础结构

在这种情况下，你需要分离DeletePropertiesHandler（参见8.5.1节），将一些功能移至一个新的DeletePropertiesOperation类。AbstractOperation超类实现了必需的IUndoableOperation接口的大部分内容。

```
public class DeletePropertiesOperation extends AbstractOperation
{
```

```
    private final PropertyElement[] elements;
```

```
    public DeletePropertiesOperation(
        PropertyElement[] elements
    ) {
        super(getLabelFor(elements));
        this.elements = elements;
    }
}
```

构造函数调用getLabelFor()方法并基于当前选中元素为操作生成可读标签。该标签在所有撤销/重做命令出现的地方出现，如在Edit菜单中。

```
private static String getLabelFor(PropertyElement[] elements) {
    if (elements.length == 1) {
        PropertyElement first = elements[0];
        if (first instanceof PropertyEntry) {
            PropertyEntry propEntry = (PropertyEntry) first;
            return "Remove property " + propEntry.getKey();
        }
        if (first instanceof PropertyCategory) {
            PropertyCategory propCat = (PropertyCategory) first;
            return "Remove category " + propCat.getName();
        }
    }
    return "Remove properties";
}
```

execute()方法提示用户以确认操作并移除指定的属性。如果info参数不是null，那么可以向它请求一个UI上下文。在该上下文中可以在运行过程中提示用户输入信息。如果监视器参数不是null，

那么它可以被用于在运行过程中向用户提供进度反馈。该方法仅当第一次执行操作时被调用。

```
public IStatus execute(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException
{
    // If a UI context has been provided,
    // then prompt the user to confirm the operation.

    if (info != null) {
        Shell shell = (Shell) info.getAdapter(Shell.class);
        if (shell != null) {
            if (!MessageDialog.openQuestion(
                shell,
                "Remove properties",
                "Do you want to remove the currently selected properties?"
            ))
                return Status.CANCEL_STATUS;
        }
    }

    // Perform the operation.

    return redo(monitor, info);
}
```

execute()方法调用redo()方法以执行实际的属性移除。该方法记录在两个额外字段中被移除的元素的信息，以使该操作可以被撤销。传递给redo()方法的参数与那些被提供给之前描述的execute()方法是完全相同的。

```
private PropertyElement[] parents;
private int[] indexes;

public IStatus redo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException
{
    // Perform the operation, providing feedback to the user
    // through the progress monitor if one is provided.

    parents = new PropertyElement[elements.length];
    indexes = new int[elements.length];

    if (monitor != null)
        monitor.beginTask("Remove properties", elements.length);

    Shell shell = (Shell) info.getAdapter(Shell.class);
    shell.setRedraw(false);
    try {
        for (int i = elements.length; --i >= 0;) {
            parents[i] = elements[i].getParent();
            PropertyElement[] children = parents[i].getChildren();
            for (int index = 0; index < children.length; index++) {
                if (children[index] == elements[i]) {
                    indexes[i] = index;
                    break;
                }
            }
        }
        elements[i].removeFromParent();
    }
```

```

        if (monitor != null)
            monitor.worked(1);
    }
}
finally {
    shell.setRedraw(true);
}

if (monitor != null)
    monitor.done();
return Status.OK_STATUS;
}

```

undo()方法通过重新向模型中插入被移除的元素反转当前操作。

```

public IStatus undo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException
{
    Shell shell = (Shell) info.getAdapter(Shell.class);
    shell.setRedraw(false);
    try {
        for (int i = 0; i < elements.length; i++) {
            if (parents[i] instanceof PropertyCategory)
                ((PropertyCategory) parents[i]).addEntry(indexes[i],
                    (PropertyEntry) elements[i]);
            else
                ((PropertyFile) parents[i]).addCategory(indexes[i],
                    (PropertyCategory) elements[i]);
        }
    }
    finally {
        shell.setRedraw(true);
    }
    return Status.OK_STATUS;
}

```

上面的undo()方法准确地将元素重新插入至模型中它们被移除的位置。这需要对PropertyCategory addEntry()方法进行一些重构 (参见8.2.3节以了解更多关于编辑器模型的内容)。

```

public void addEntry(PropertyEntry entry) {
    addEntry(entries.size(), entry);
}

public void addEntry(int index, PropertyEntry entry) {
    if (!entries.contains(entry)) {
        entries.add(index, entry);
        ((PropertyFile) getParent()).entryAdded(
            this, entry);
    }
}

```

以下是对PropertyFile addCategory()方法做出的类似重构。

```

public void addCategory(PropertyCategory category) {
    addCategory(categories.size(), category);
}

public void addCategory(int index, PropertyCategory category) {
    if (!categories.contains(category)) {

```

```
        categories.add(index, category);
        categoryAdded(category);
    }
}
```

不移除选中属性，DeletePropertiesHandler现在必须创建一个将被移除的属性的数组，并稍后传递至一个DeletePropertiesOperation的实例。该操作与一个用于提示用户和进度监视器输入用户反馈的UI上下文一起被传递至编辑器的撤销/重做管理程序以用于执行。如果在运行过程中发生了异常，你可以使用ExceptionsDetailsDialog（参见11.1.9节）而不是下面的MessageDialog。

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (!(selection instanceof IStructuredSelection))
        return null;
    final IEditorPart editor = HandlerUtil.getActiveEditor(event);
    if (!(editor instanceof PropertiesEditor))
        return null;
    return execute(
        (PropertiesEditor) editor, (IStructuredSelection) selection);
}

private Object execute(
    final PropertiesEditor editor, IStructuredSelection selection
) {

    // Build an array of properties to be removed.
    Iterator<?> iter = selection.iterator();
    int size = selection.size();
    PropertyElement[] elements = new PropertyElement[size];
    for (int i = 0; i < size; i++)
        elements[i] = (PropertyElement) ((Object) iter.next());

    // Build the operation to be performed.
    DeletePropertiesOperation op
        = new DeletePropertiesOperation(elements);
    op.addContext(editor.getUndoContext());

    // The progress monitor so the operation can inform the user.
    IProgressMonitor monitor = editor.getEditorSite().getActionBars()
        .getStatusLineManager().getProgressMonitor();

    // An adapter for providing UI context to the operation.
    IAdaptable info = new IAdaptable() {
        public Object getAdapter(Class adapter) {
            if (Shell.class.equals(adapter))
                return editor.getSite().getShell();
            return null;
        }
    };

    // Execute the operation.
    try {
        editor.getOperationHistory().execute(op, monitor, info);
    }
    catch (ExecutionException e) {
        MessageDialog.openError(editor.getSite().getShell(),
```



```

        "Remove Properties Error",
        "Exception while removing properties: " + e.getMessage());
    }
    return null;
}

```

上面的execute方法调用PropertiesEditor中的一些新方法。

```

public IOperationHistory getOperationHistory() {
    // The workbench provides its own undo/redo manager
    //return PlatformUI.getWorkbench()
    //    .getOperationSupport().getOperationHistory();
    // which, in this case, is the same as the default undo manager
    return OperationHistoryFactory.getOperationHistory();
}

public IUndoContext getUndoContext() {
    // For workbench-wide operations, we should return
    //return PlatformUI.getWorkbench()
    //    .getOperationSupport().getUndoContext();
    // but our operations are all local, so return our own content
    return undoContext;
}

```

该undoContext方法必须在一个新的initUndoRedo()方法中与撤销和重做命令一起被初始化。该initUndoRedo()方法从createPages()方法中被调用。

```

private UndoActionHandler undoAction;
private RedoActionHandler redoAction;
private IUndoContext undoContext;

private void initUndoRedo() {
    undoContext = new ObjectUndoContext(this);
    undoAction = new UndoActionHandler(getSite(), undoContext);
    redoAction = new RedoActionHandler(getSite(), undoContext);
}

```

这些新的撤销和重做操作应出现于上下文菜单中，因此修改fillContextMenu()方法。

```

private void fillContextMenu(IMenuManager menuMgr) {
    menuMgr.add(undoAction);
    menuMgr.add(redoAction);
    menuMgr.add(new Separator());
    ... existing code ...
}

```

然后，修改pageChange()方法以调用两个新的方法，这样新的撤销和重做操作将会被关联至Edit菜单中的全局撤销和重做操作。

```

protected void pageChange(int newPageIndex) {
    switch (newPageIndex) {
        case 0 :
            if (isDirty())
                updateTreeFromTextEditor();
            setTreeUndoRedo();
            break;
        case 1 :
            if (isPageModified)

```

```
        updateTextEditorFromTree();
        setTextEditorUndoRedo();
        break;
    }
    isPageModified = false;

    super.pageChange(newPageIndex);
}

private void setTreeUndoRedo() {
    final IActionBars actionBars = getEditorSite().getActionBars();

    actionBars.setGlobalActionHandler(ActionFactory.UNDO.getId(), undoAction);
    actionBars.setGlobalActionHandler(ActionFactory.REDO.getId(), redoAction);

    actionBars.updateActionBars();
}

private void setTextEditorUndoRedo() {
    final IActionBars actionBars = getEditorSite().getActionBars();

    IAction undoAction2 = textEditor.getAction(ActionFactory.UNDO.getId());
    actionBars.setGlobalActionHandler(ActionFactory.UNDO.getId(), undoAction2);

    IAction redoAction2 = textEditor.getAction(ActionFactory.REDO.getId());
    actionBars.setGlobalActionHandler(ActionFactory.REDO.getId(), redoAction2);

    actionBars.updateActionBars();
}
```

最后，源代码页面的撤销/重做栈与属性页面的撤销/重做栈分离开来，因此，添加以下行代码至setTextEditorUndoRedo()方法的末尾以在页面改变时清理撤销/重做栈。

```
getOperationHistory().dispose(undoContext, true, true, false);
```

一个更好的解决办法与已讨论过的类似，但没有在这里实现。它将和这两个撤销/重做栈合并为一个由Properties和Source页面共享的统一撤销/重做栈。

如果操作共享通用撤销上下文，同时也具有一些没有被共享的上下文，那么存在来源于一个上下文的操作将会在一个线性样式中被撤销。然而，在另一个上下文中的操作可能会被跳过。要减少这种问题的发生，你可以注册一个IOperationApprover的实例以确保在所有更高优先级的操作被撤销之后才会撤销该操作。

该接口还提供了确认影响活动编辑器外部的上下文并且不会立即向用户显示的撤销和重做操作的方法。Eclipse平台包含了IOperationApprover的下列子类。这些子类在使用重叠上下文管理撤销/重做操作是十分有用的。

- **LinearUndoEnforcer**——一个强制使用严格的线性撤销的操作验证程序。它不允许对任意不是在其所有撤销上下文中最新可用的操作进行撤销或重做。
- **LinearUndoViolationUserApprover**——提示用户查看是否允许线性撤销冲突的操作验证程序。当一个正被撤销或重做的操作与历史记录中时间更新的另一个操作共享撤销上下文时，线性撤销冲突将会被检测到。
- **NonLocalUndoUserApprover**——一个提示用户查看一个非本地撤销是否应在编辑器内部进行的操作验证程序。当正在被撤销或重做的操作影响除了那些被编辑器自身描述之外的操作，

将会检测到非本地撤销。

Eclipse SDK包含了一个基本的撤销/重做实例作为eclipse-examples-3.4-win32.zip的一部分。它提供了在这里没有被讨论到的额外的撤销/重做代码，如UndoHistoryView和IOperationApprover的实现。它们用于在操作历史记录中验证特定操作的撤销和重做。

8.5.5 剪贴板操作

编辑器的基于剪贴板的操作对于它们的各自的基于视图的操作是一样的（参见7.3.7节）。

8.6 链接编辑器

活动的编辑器中的选择可以使用与链接视图选择类似的技术被链接至它外围的视图（参见7.4节）。此外，编辑器可以通过实现getAdapter(Class)方法为大纲视图提供内容，与以下类似（参见7.4.2节以了解更多关于适配器的内容）：

```
private PropertiesOutlinePage outlinePage;

public Object getAdapter(Class adapter) {
    if (adapter.equals(IContentOutlinePage.class)) {
        if (outlinePage == null)
            outlinePage = new PropertiesOutlinePage();
        return outlinePage;
    }
    return super.getAdapter(adapter);
}
```

一般地，PropertiesOutlinePage类通过扩展ContentOutlinePage和实现一些方法实现了IContentOutlinePage。这些方法与本章和上一章中讨论的方法类似，因此，在这里不会对它们进行细节讨论。

8.7 RFRS相关事项

《RFRS Requirements》的“用户界面”一节包含了11个（5个要求和6个最佳做法）与编辑器相关的内容。它们都是来源于Eclipse UI准则。

8.7.1 使用编辑器进行编辑或浏览（RFRS 3.5.9）

用户界面准则#6.1是一个要求，它说明：

使用编辑器编辑或浏览文件、文档或其他输入对象。该要求测试编辑器被用于编辑器文件（或类似输入）和视图被用于辅助导航（如导航程序）或处理简单的修改任务（如属性视图）。编辑器必须从资源中双击或单击（取决于工作台的单击行为参数）以打开。由于视图可以使用编辑器同时打开，当输入遵守打开-保存-关闭生命周期时只打开一个视图是不合适的。视图不能重定向编辑器操作，向通用工具栏添加项，或替代合适编辑器行为和外观。

对于该测试，你需要说明由你的插件提供的编辑器。显示文件、文档或其他输入类型是如何被用于编辑或浏览的。基于本章所展示的示例，显示Property File编辑器是如何被用于编辑属性文件的。

8.7.2 编辑器生命周期（RFRS 3.5.10）

用户界面准则#6.2是一个要求，它说明：

编辑器中做出的更改必须遵循打开-保存-关闭生命周期模型。当第一次打开一个编辑器时，编辑器的内容应当是未修改过的（干净的）。如果内容被修改过，编辑器就应该更改与平台进行交流。作为回应，编辑器的选项卡上将出现一个星号（*）。更改应缓存于编辑模型中直到用户明确保存它们。在那时，更改应被提交至模型存储。

为了通过这项测试，在一个文件上打开你的编辑器并显示编辑器一开始是未修改过的。然后，使用编辑器做出一个更改并显示在编辑器的选项卡中出现了星号（*）。最后，保存编辑器并显示更改已经被提交至文件，并且编辑器返回了它的未修改状态。参见8.4节以了解更多关于编辑器生命周期的信息。

8.7.3 访问全局操作 (RFRS 3.5.11)

用户界面准则#6.9是一个要求，它说明：

如果一个编辑器支持cut、copy、paste或任何全局操作，与窗口菜单和工具栏中的操作相同的操作必须是可执行的。窗口菜单包含一些全局操作，如Edit菜单中的cut、copy和paste。这些操作的目标是活动部分，如同阴影标题区域所表示的那样。如果在一个编辑器中支持这些操作，该编辑器应关联这些窗口操作以使窗口菜单或工具栏中的选择产生与编辑器中对应操作的同样效果。编辑器不应忽略这些操作，并且应将操作复制至窗口菜单或工具栏。以下是受支持的全局操作：

- a. 撤销
- b. 重做
- c. 剪切
- d. 复制
- e. 粘贴
- f. 打印
- g. 删除
- h. 查找
- i. 全选
- j. 设置书签

显示你的编辑器支持所有相关的全局操作，如cut、copy和paste。从你的编辑器内部触发这些操作，然后展示它们也可以从工作台的菜单栏被触发，并具有同样的效果。对于Properties编辑器来说，你应展示全局操作，如删除，可以从编辑器的Properties和Source页面内部被访问。参见8.5.2节以了解更多关于关联全局操作的内容。

8.7.4 当对象被删除时关闭 (RFRS 3.5.12)

用户界面准则#6.16是一个要求，它说明：

如果一个编辑器的输入被删除并且编辑器不包含更改，编辑器应被关闭。当从导航程序之一（如导航视图、J2EE视图、数据视图或SDP中的DBA资源管理器视图）删除资源时，对于当前在该资源上打开的所有编辑器的处理取决于编辑器是否具有未保存的更改。如果编辑器在资源上一次被保存后不包含任意更改，那么编辑器应立即被关闭。

展示你的编辑器当它的一个输入对象（如一个指定资源）被删除时将自动被关闭。对于Properties编辑器而言，你应创建一个新的属性文件，并使用Properties编辑器打开它，然后从Navigator视图中删除该文件。如果你通过使用本章描述的编辑器框架实现你的编辑器，框架应自动强制实现这一准则。

8.7.5 同步外部更改 (RFRS 3.5.14)

用户界面准则#6.30是一个要求,它说明:

如果对于资源做出的更改是在工作台外部做出的,用户应被提示覆盖这些在工作台外部做出的更改,或收回当保存操作在编辑器中被触发时的Save操作。

在一个文件上打开你的编辑器并做出一个更改。然后,在Eclipse外部更改该文件。最后,切换回Eclipse并视图保存该文件。展示你被提示覆盖外部更改或取消保存操作。如果你使用本章中描述的编辑器框架来实现编辑器,则该框架应自动强制实现这一准则。

8.7.6 注册编辑器菜单 (RFRS 5.3.5.2)

用户界面准则#6.14是一个最佳做法,它说明:

在编辑器中注册所有的Eclipse上下文菜单。在Eclipse中,编辑器的菜单和工具栏将自动被平台扩展。相反的是,编辑器和Eclipse的结合支持上下文菜单扩展项。为了获取该结合,编辑器在Eclipse中注册所有它包含的上下文菜单。

为了通过该项测试,展示编辑器的上下文菜单已经被注册至Eclipse。如果它们正确地被注册,你应看到系统添加合适的上下文菜单项。参见8.5.1节以了解更多关于上下文菜单的内容。

8.7.7 编辑器操作过滤器 (RFRS 5.3.5.3)

用户界面准则#6.15是一个最佳做法。它说明:

为编辑器中每一种对象类型实现一个操作过滤器。操作过滤器使得一个插件向由另一个插件定义的编辑器中的对象添加操作变得容易。

对于这项测试,展示菜单操作过滤可以在你的编辑器编辑的对象上发挥作用。参见6.7.2节以了解更多关于创建可以被其他插件扩展的上下文菜单的内容。

8.7.8 未保存的编辑器更改 (RFRS 5.3.5.4)

用户界面准则#6.17是一个最佳做法。它说明:

如果编辑器的输入被删除并且编辑器包含更改,编辑器应给用户一个保存更改至另一个位置的机会,然后再关闭。

我们从用编辑器打开一个文件并做出一个更改开始。然后,在Navigator视图中选择该文件并删除它。展示显示一个警告信息,通知用户编辑器包含未保存的更改。为了通过该准则的最佳做法部分,用户应被提供一个选项以保存文件至另一个位置。如果你使用本章中描述的编辑器框架来实现你的编辑器,框架应自动强制使用该准则。

8.7.9 为更改过的资源添加前缀 (RFRS 5.3.5.5)

用户界面准则#6.18是一项最佳做法。它说明:

如果一个资源是被更改过的,在编辑器选项卡中显示的资源名称前添加一个星号。

这本质上是准则#14的一个子集。在你的编辑器中编辑一个文件并展示文件名前被添加了一个星号。如果你使用本章中描述的编辑器框架来实现编辑器,框架应自动强制实现该准则。

8.7.10 编辑器大纲视图 (RFRS 5.3.5.6)

用户界面准则#6.20是一个最佳做法。它说明:

如果编辑器中的数据太多而不能在一个屏幕中显示,并将折叠为一个结构化的大纲,编辑器应为大纲视图提供一个大模型。在Eclipse中,在每一个编辑器和大纲视图之间有一个特别的关系。当打开一个编辑器时,大纲视图将会连接至该编辑器并向它请求一个大模型。如果编辑器返回了一个大模型,那么无论何时编辑器是活动的,该模型都将显示于大纲视图中。大纲被用于在编辑数据间导航,或在一个更高层次的抽象上与编辑数据交互。

对于这项测试,打开你的编辑器并显示它更新Outline视图的内容和允许导航数据结构。如果你编辑器的一个不同的实例被选中,展示Outline视图的内容也恰当地更改。参见8.4节以了解关于链接编辑器至Outline视图的信息。

8.7.11 与大纲视图同步 (RFRS 5.3.5.7)

用户界面准则#6.21是一个最佳做法。它说明:

关于编辑器和大纲视图间位置的通知应当是双向的。在合适的时候,上下文菜单应在大纲视图中可用。

选择Outline视图中的一项并展示它选择了编辑器中的对应项。然后,在编辑器中选择一项并展示它选择了Outline视图中的对应项。

8.8 总结

本章说明了关于如何创建新编辑器以用于编辑和浏览工作台资源的编辑器的细节。它展示了如何建立一个多页面的编辑器,处理编辑器的生命周期,并创建不同的编辑器操作。

参考文献

本书资源(2.9节).

Deva, Prashant, "Folding in Eclipse Text Editors," March 11, 2005(www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html).

Ho, Elwin, "Creating a Text-Based Editor for Eclipse," HP, June 2003 (devresource.hp.com/drc/technical_white_papers/eclipteditor/index.jsp).



第9章 资源更改跟踪

Eclipse系统生成资源更改事件。这些事件表示诸如在操作过程中已经被添加、修改和移除的文件和文件夹等。感兴趣的对象可以订阅这些事件并采取任何必需的操作以将它们和Eclipse保持同步。

为了说明资源更改跟踪，我们将会修改Favorites视图（参见第7章），使得无论何时资源被删除时，你可以从Favorites视图中移除对应元素。

9.1 IResourceChangeListener

Eclipse使用接口org.eclipse.core.resources.IResourceChangeListener在资源改变时通知注册的监听器。FavoritesManager（参见7.2.3节）需要将它的Favorites项列表与Eclipse保持同步。这项功能通过实现org.eclipse.core.resources.IResourceChangeListener接口并注册资源更改事件来实现。

此外，必须修改FavoritesActivator stop()方法以调用新的FavoritesManager shutdown()方法。这样，当插件被关闭后，资源更改将不再通知管理程序。现在，无论何时发生资源更改，Eclipse将调用resourceChanged()方法。

```
public class FavoritesManager
    implements IResourceChangeListener
{
    private FavoritesManager() {
        ResourcesPlugin.getWorkspace().addResourceChangeListener (
            this, IResourceChangeEvent.POST_CHANGE);
    }

    public static void shutdown(){
        if (manager != null){
            ResourcesPlugin.getWorkspace()
                .removeResourceChangeListener(manager);
            manager.saveFavorites();
            manager = null;
        }
    }

    public void resourceChanged(IResourceChangeEvent e){
        //Process events here.
    }
    ... existing code from Section 7.2.3, View model, on page 295 ...
}
```

9.1.1 IResourceChangeEvent

FavoritesManager仅对已经发生的更改感兴趣，应此它在订阅更改事件时使用了IResourceChangeEvent.POST_CHANGE常量。Eclipse提供了几个可以联合起来用于指定应何时将资源更改通知感兴趣的对象的IResourceChangeEvent常量。下面是一个合法的常量列表。它们也出现于IResourceChangeEvent的Javadoc中。

- **PRE_BUILD**——事先报告构建器的活动（参见14.1节）。
 - **PRE_CLOSE**——事先报告由getResource()返回的单个项目即将关闭。
 - **PRE_DELETE**——事先报告由getResource()返回的单个项目即将删除。
 - **PRE_REFRESH**——事先报告项目刷新。
 - **POST_BUILD**——事后报告构建器的活动（参见14.1节）。
 - **POST_CHANGE**——事后报告一个或多个资源的创建、删除和修改。这些活动表现为由getDelta()返回的层次结构的资源增量。
- IResourceChangeEvent接口还定义了一些可以用于查询它状态的方法。
- **findMarkerDeltas(String, boolean)**——返回指定类型的所有标记增量。该指定类型与该事件的资源增量相关联。如果你想要包括指定类型的子类型，将true作为第二个参数传递。
 - **getBuildKind()**——返回导致该事件的构建的类型。
 - **getDelta()**——返回位于工作区的资源增量。该资源增量描述了工作区中的资源发生的更改的集合。
 - **getResource()**——返回被请求的资源。
 - **getSource()**——返回指定该事件源的对象。
 - **getType()**——返回正在被报告的事件的类型。

9.1.2 IResourceDelta

每一个独立的更改都被编码为一个资源增量的实例。该资源增量通过IResourceDelta接口表示。Eclipse提供了几个一起用于指定由系统处理的资源增量的常量。下面是合法常量的列表。它们也出现于IResourceDelta的Javadoc中。

- **ADDED**——表示该资源已经被添加至它的父辈的增量类型常量。
- **ADDED_PHANTOM**——表示影子资源已经被添加至增量代码的位置的增量类型常量。
- **ALL_WITH_PHANTOMS**——描述所有可能的增量类型的位掩码，包括那些相关的影子资源。
- **CHANGED**——表示资源已经被修改的增量类型常量。
- **CONTENT**——表示资源内容已经更改的更改常量。
- **COPIED_FROM**——表示资源从另一个位置被复制的更改常量。
- **DESCRIPTION**——表示项目的描述已经更改的更改常量。
- **ENCODING**——表示资源的编码已经更改的更改常量。
- **LOCAL_CHANGED**——表示链接资源的底层文件或文件夹已经被添加或移除的更改常量。
- **MARKERS**——表示资源的标记已经更改的更改常量。
- **MOVED_FROM**——表示资源从另一个位置被移除的更改常量。
- **MOVED_TO**——表示资源被移至另一位置的更改常量。
- **NO_CHANGE**——表示资源还未发生任何更改的增量类型常量。
- **OPEN**——表示资源被打开或关闭的更改常量。
- **REMOVED**——表示资源已经从它的父辈移除的增量类型常量。
- **REMOVED_PHANTOM**——表示影子资源已经从增量代码的位置移除的增量类型常量。
- **REPLACED**——表示资源已经被另一个资源在同一个位置所取代的更改常量（比如，资源已经被删除，然后又被添加）。

- SYNC——表示资源的同步状态已经更改的更改常量。
- TYPE——表示资源的类型已经更改的更改常量。

IResourceDelta类还定义了很多常用API，包括：

- accept(IResourceDeltaVisitor)——访问被添加（ADDED）、更改（CHANGED）或移除（REMOVED）的资源增量。如果访问者返回true，资源增量的后代也被访问。
- accept(IResourceDeltaVisitor, boolean)——与上面的方法类似，但可以选择是否包含影子资源。
- accept(IResourceDeltaVisitor, int)——与上面的方法类似，但可以选择是否包含影子资源和/或组私有成员。
- findMember(IPath)——查找并返回由该增量中的给定路径所指定的后代增量。如果不存在这样的后代，则返回null。
- getAffectedChildren()——为当前被添加（ADDED）、更改（CHANGED）或移除（REMOVED）的资源增量的所有后代返回资源增量。
- getAffectedChildren(int)——为指定掩码中包含的种类型的资源的所有后代返回资源增量。
- getFlags()——返回更多细节描述资源是如何被影响的标志。
- getFullPath()——返回该资源增量的完整绝对路径。
- getKind()——返回该资源增量的种类。
- getMarkerDeltas()——返回对应资源的标记的更改。
- getMovedFromPath()——返回该资源（在“之后”状态）被移除的位置的完整路径（在“之前”状态）。
- getMovedToPath()——返回该资源（在“之前”状态）被移至的位置的完整路径（在“之后”状态）。
- getProjectRelativePath()——返回该资源增量的相对于项目的路径。
- getResource()——返回受影响资源的句柄。

9.2 处理更改事件

POST_CHANGE资源更改事件不是表示为单个更改，而是表示为描述一个或多个已经发生的更改的层次结构。出于对效率的考虑，以这种方式将事件分批。将发生的每一个更改报告给每一个感兴趣的对象将显著减慢系统的速度并降低对用户的响应性。要看到该更改层次结构，添加以下代码至FavoritesManager。

```
public void resourceChanged(IResourceChangeEvent event) {
    System.out.println(
        "FavoritesManager - resource change event");
    try {
        event.getDelta().accept(new IResourceDeltaVisitor() {
            public boolean visit(IResourceDelta delta)
                throws CoreException
            {
                StringBuffer buf = new StringBuffer(80);
                switch (delta.getKind()) {
                    case IResourceDelta.ADDED:
                        buf.append("ADDED");
                        break;
                    case IResourceDelta.REMOVED:
```

```

        buf.append("REMOVED");
        break;
    case IResourceDelta.CHANGED:
        buf.append("CHANGED");
        break;
    default:
        buf.append("[");
        buf.append(delta.getKind());
        buf.append("]");
        break;
    }
    buf.append(" ");
    buf.append(delta.getResource());
    System.out.println(buf);
    return true;
}
});
}
catch (CoreException ex) {
    FavoritesLog.logError(ex);
}
}
}

```

上面的代码将生成描述系统中的资源更改的层次结构的文本表示。要查看这部分代码的执行结果，启动Runtime Workbench（参见2.6节）并打开Favorites视图。在Runtime Workbench中，创建一个简单的项目，然后添加如下所示的文件夹和文件（图9-1）。

在创建过程中，你将会看到生成至Console视图的输出。这些输出描述了由Eclipse传送的资源更改事件。FavoritesManager对资源的删除尤其感兴趣。但你删除这两个文件时，你将在Console看到以下内容：

```

FavoritesManager - resource change event
CHANGED R/
CHANGED P/Test
CHANGED F/Test/folder1
CHANGED F/Test/folder1/folder2
REMOVED L/Test/folder1/folder2/file1.txt
REMOVED L/Test/folder1/folder2/file2.txt

```

接下来的步骤是修改FavoritesManager方法以使用该信息完成一些任务。更改将使得FavoritesManager可以移除Favorites项。这些Favorites项引用了已经从系统移除的资源。

```

public void resourceChanged(IResourceChangeEvent event) {
    Collection<IFavoriteItem> itemsToRemove
        = new HashSet<IFavoriteItem>();
    try {
        event.getDelta().accept(new IResourceDeltaVisitor() {
            public boolean visit(IResourceDelta delta)
                throws CoreException
            {
                if (delta.getKind() == IResourceDelta.REMOVED) {
                    IFavoriteItem item =
                        existingFavoriteFor(delta.getResource());
                    if (item != null)
                        itemsToRemove.add(item);
                }
            }
        });
    }
}

```

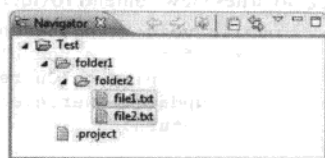


图9-1 导航视图

```

        }
        return true;
    }
    });
}
catch (CoreException ex) {
    FavoritesLog.logError(ex);
}
if (itemsToRemove.size() > 0)
    removeFavorites(itemsToRemove.toArray(
        new IFavoriteItem[itemsToRemove.size()]));
}

```

当完成了上面的代码后，启动Runtime Workbench测试该更改。这项测试在项目中创建一个文件，将该文件作为一个Favorites项添加至Favorites视图，然后从项目中删除该文件。文件被移除了，但Favorites项不会像预期的那样被移除。查看.log文件（参见3.6.2节）将发现以下异常：

```
org.eclipse.swt.SWTException: Invalid thread access
```

这表示一个SWT组件，比如收藏夹视图中的表，正在被一个UI线程之外的线程访问（参见4.2.5节以了解更多关于Display.getDefault()和UI线程的内容）。要减少这种问题的发生，根据以下内容修改FavoritesViewContentProvider favoritesChanged()方法以保证查看器是被UI线程访问。

```

public void favoritesChanged(final FavoritesManagerEvent event) {
    // If this is the UI thread, then make the change.
    if (Display.getCurrent() != null) {
        updateViewer(event);
        return;
    }

    // otherwise, redirect to execute on the UI thread.
    Display.getDefault().asyncExec(new Runnable() {
        public void run() {
            updateViewer(event);
        }
    });
}

private void updateViewer(final FavoritesManagerEvent event) {
    // Use the setRedraw method to reduce flicker
    // when adding or removing multiple items in a table.
    viewer.getTable().setRedraw(false);
    try {
        viewer.remove(event.getItemsRemoved());
        viewer.add(event.getItemsAdded());
    }
    finally {
        viewer.getTable().setRedraw(true);
    }
}
}

```

9.3 批处理更改事件

当任何UI插件修改的资源时，它应通过继承org.eclipse.ui.actions.WorkspaceModifyOperation封装资源修改代码。使用该操作的主要后果是一般作为工作区更改的结果（比如，对资源增量的触发，

执行自动构建等)而发生的事件将被延迟直到最外层的操作已经成功完成之后发生。在Favorites视图中,如果你想要实现一个本身删除底层资源的删除操作,而不是仅删除引用该资源的Favorites项,那么它可以根据以下所示来实现。

从WorkspaceModifyOperation继承并被Action或IActionDelegate调用的run()方法,首先调用execute()方法,然后触发一个更改事件。该事件包含了所有由execute()方法更改的资源。

```
package com.qualityeclipse.favorites.actions;

import ...

public class DeleteResourcesOperation
    extends WorkspaceModifyOperation
{
    private final IResource[] resources;

    public DeleteResourcesOperation(IResource[] resources) {
        this.resources = resources;
    }

    protected void execute(IProgressMonitor monitor)
        throws
            CoreException,
            InvocationTargetException,
            InterruptedException
    {
        monitor.beginTask("Deleting resources...", resources.length);
        for (int i = 0; i < resources.length; i++) {
            if (monitor.isCanceled())
                break;
            resources[i].delete(
                true, new SubProgressMonitor(monitor, 1));
        }
        monitor.done();
    }
}
```

如果你在一个没有头部的Eclipse环境中,或是在一个不依赖任何UI插件的插件中修改资源,WorkspaceModifyOperation类是不可访问的。在这种情况下,使用IWorkspace.run()方法批处理更改事件。

```
protected void execute(IProgressMonitor monitor)
    throws CoreException
{
    ResourcesPlugin.getWorkspace().run(new IWorkspaceRunnable() {
        public void run(IProgressMonitor monitor) throws CoreException
        {
            monitor.beginTask(
                "Deleting resources...", resources.length);
            for (int i = 0; i < resources.length; i++) {
                resources[i].delete(
                    true, new SubProgressMonitor(monitor, 1));
            }
            monitor.done();
        }
    }, monitor);
}
```

9.4 进度监视器

对于长运行时间的操作而言，进度监视器显示当前正进行的操作和一个对还剩下要完成的任务的估计。在上面的代码中，一个进度监视器用于与用户交流，显示资源正被删除和还有多少资源在操作完成前需要被删除（参见以上几节的方法和8.5.4节中的redo()方法）。

此外，由于DeleteResourcesOperation与用户界面交互，将周期性地调用isCanceled()查看用户是否已经取消该操作。没有什么比看着一个具有取消按钮的长运行时间的操作仅仅是为了查证取消按钮有没有作用更令人沮丧的事情了。

9.4.1 IProgressMonitor

org.eclipse.core.runtime.IProgressMonitor接口提供了方法用于显示操作开始的时间，有多少已经完成，和它何时完成。

- beginTask(String, int)——由操作调用一次以显示该操作已经开始和在它完成之前有多少工作需要做。该方法应当仅由进度监视器的每一个实例调用一次。
- done()——由操作调用以显示它完成。
- isCanceled()——操作应定期轮询该方法以查看是否用户已经请求取消该操作。
- setCanceled(boolean)——该方法一般当用户在操作过程中点击取消按钮时，由UI代码调用，将取消状态设置为true。
- setTaskName(String)——设置向用户显示的任务的名称。通常，并没有什么需要调用该方法。因为任务名称通过beginTask(String, int)设置。
- worked(int)——由操作调用，用于表示指定数目的工作单元已经被完成。

IProgressMonitorWithBlocking接口为监视器扩展了IProgressMonitor。这些监视器想要在活动由于另一线程中的并行活动而被阻止时支持反馈。如果一个运行操作在某些时候调用下面列出的setBlocked方法，它必须在操作完成之前最后调用clearBlocked方法。

- clearBlocked()——由操作调用以显示操作不再被阻止了。
- setBlocked(IStatus)——由操作调用以显示该操作被一些后台活动所阻止。

9.4.2 用于显示进度的类

Eclipse提供了几个类。这些类或实现IProgressMonitor接口，或通过IRunnableWithProgress接口提供一个进度监视器。这些类在不同情况下使用以将长运行时操作的进度通知给用户。

- SubProgressMonitor——由一个父辈操作传递给子操作的进度监视器，以使子操作可以向用户通知作为父操作的一部分的进度（参见9.3节）。
- NullProgressMonitor——一个支持取消但不提供任何用户反馈的进度监视器。它适用于继承。
- ProgressMonitorWrapper——封装另一个进度监视器的进度监视器。它转发IProgressMonitor和IProgressMonitorWithBlocking方法至被封装的进度监视器。它适用于继承。
- WorkspaceModifyOperation——一个批处理资源更改事件的操作。它还提供了作为它执行的一部分的一个进度监视器（参见9.3节）。
- ProgressMonitorPart——一个SWT复合组件。它由一个显示任务和子任务名称的标签，以及一个显示进度的进度显示器组成。
- ProgressMonitorDialog——打开一个向用户显示进度的对话框，并提供一个操作使用的进度监

视器以用于引用该信息。

- `TimeTriggeredProgressMonitorDialog`——在操作执行时等待指定长度的时间，然后打开一个向用户显示进度的对话框并提供一个操作使用的进度监视器以用于引用该信息。如果操作在指定长度的时间之前完成了，那么将不会打开任何对话框。这是一个内部工作台类，但我们在这里列出它。这是因为它所表达的观念和它的功能是令人感兴趣的。为了了解更多内容，参见9.4.4节和位于bugs.eclipse.org/bugs/show_bug.cgi?id=123797的Bugzilla条目123797。
- `WizardDialog`——当打开时，可以选择是否提供进度信息以作为向导的一部分。该向导实现了 `IRunnableContext`，因此该操作可以调用 `run(boolean, boolean, RunnableWithProgress)` 并通过提供的进度监视器在向导中显示进度（参见11.2.3节和11.2.6节）。

作为一个示例，我们实现了一个新的 `DeleteResourcesHandler` 以打开一个 `ProgressMonitorDialog`，然后运行 `DeleteResourcesOperation` 以执行实际操作。新的 `DeleteResourcesHandler` 类扩展了 `AbstractHandler`，并包含一个与以下类似的 `execute` 方法：

```
public Object execute(ExecutionEvent event)
    throws ExecutionException
{
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (!(selection instanceof IStructuredSelection))
        return null;
    IStructuredSelection structSel = (IStructuredSelection) selection;
    // Build a collection of selected resources
    final Collection<IResource> resources = new HashSet<IResource>();
    for (Iterator<?> iter = structSel.iterator(); iter.hasNext();) {
        Object element = iter.next();
        if (element instanceof IAdaptable)
            element = ((IAdaptable) element).getAdapter(IResource.class);
        if ((element instanceof IResource))
            resources.add((IResource) element);
    }
    // Execute the operation
    try {
        // Display progress either using the ProgressMonitorDialog ...
        //Shell shell =HandlerUtil.getActiveShell(event);
        //IRunnableContext context =new ProgressMonitorDialog(shell);
        // ... or using the window's status bar ...
        //IWorkbenchWindow context
        // =HandlerUtil.getActiveWorkbenchWindow(event);
        // ... or using the workbench progress service
        IWorkbenchWindow window
            =HandlerUtil.getActiveWorkbenchWindow(event);
        IRunnableContext context
            =window.getWorkbench().getProgressService();
        context.run(true,false,new IRunnableWithProgress(){
            public void run(IProgressMonitor monitor)
                throws InvocationTargetException,InterruptedException
            {
                new DeleteResourcesOperation(resources.toArray(
```

```

        new IResource [resources.size()])).run(monitor);
    }
    });
}
catch (Exception e) {
    FavoritesLog.logError(e);
}
return null;
}
}

```

9.4.3 工作台窗口状态栏

工作台窗口在窗口的底部提供了一个进度显示区域。使用IWorkbenchWindow.run()方法执行操作，并且传递给IRunnableWithProgress的进度监视器将会是状态中的进度监视器。比如，一个处理器的如下代码片段（参见6.3节以了解更多关于创建处理器的内容）显示了状态栏中的模拟进度：

```

public Object execute(ExecutionEvent event)
    throws ExecutionException
{
    IWorkbenchWindow window
        = HandlerUtil.getActiveWorkbenchWindow(event);
    try {
        window.run(true, true, new IRunnableWithProgress() {
            public void run(IProgressMonitor monitor)
                throws InvocationTargetException, InterruptedException {
                monitor.beginTask("simulate status bar progress:", 20);
                for (int i = 20; i > 0; --i) {
                    monitor.subTask("seconds left = " + i);
                    Thread.sleep(1000);
                    monitor.worked(1);
                }
                monitor.done();
            }
        });
    }
    catch (InvocationTargetException e) {
        FavoritesLog.logError(e);
    }
    catch (InterruptedException e) {
        // User canceled the operation... just ignore.
    }
}

```

如果你有一个视图或编辑器，你可以通过IWorkbenchPart获取所包含的IWorkbenchWindow。IViewPart和IEditorPart都继承了该IWorkbenchPart。

```

IWorkbenchWindow window = viewOrEditor
    .getSite().getWorkbenchWindow();

```

你也可以直接通过IStatusLineManager接口获取状态栏中的进度监视器。

```

viewPart.getViewSite().getActionBars()
    .getStatusLineManager().getProgressMonitor()

```

或

```

editorPart.getEditorSite().getActionBars()
    .getStatusLineManager().getProgressMonitor()

```

9.4.4 IProgressService

另一种用于在工作台中显示进度的机制使用了IProgressService接口。IWorkbenchWindow中的run()方法在状态栏中显示进度，而IProgressService接口使用一个名为TimeTriggeredProgress Monitor Dialog的ProgressMonitorDialog的子类显示进度。尽管你可以使用一个ProgressMonitorDialog，但如果操作需要执行超过一个指定长度的时间（在这里是800毫秒），IProgressService仅打开一个进度对话框。

```
window.getWorkbench().getProgressService().run(true, true,
    new IRunnableWithProgress() {
        public void run(IProgressMonitor monitor)
            throws InvocationTargetException, InterruptedException
        {
            monitor.beginTask("Simulated long running task #1", 60);
            for (int i = 60; i > 0; --i) {
                monitor.subTask("seconds left = " + i);
                if (monitor.isCanceled()) break;
                Thread.sleep(1000);
                monitor.worked(1);
            }
            monitor.done();
        }
    });
```

一般地，任务是在后台执行（参见21.8节），但IProgressService提供了showInDialog()方法和UIJob类用于在前台执行它们。

9.5 被延迟的更改事件

Eclipse使用惰性初始化——仅当需要时才载入插件。惰性初始化带来了一个问题，就是插件需要跟踪更改。当插件没有被载入时，该如何跟踪更改？

Eclipse通过为一个未被载入的插件排序更改事件解决了该问题。当该插件被载入时，它接受了一个单独资源更改事件。该事件包含了所有在插件非活动时发生的更改的集合。要接受该事件，你的插件必须在它启动时，注册为一个保存参与者，如下所示：

```
public static void addSaveParticipant() {
    ISaveParticipant saveParticipant = new ISaveParticipant(){
        public void saving(ISaveContext context)
            throws CoreException
        {
            // Save any model state here.
            context.needsDelta();
        }
        public void doneSaving(ISaveContext context) {}
        public void prepareToSave(ISaveContext context)
            throws CoreException {}
        public void rollback(ISaveContext context) {}
    };

    ISavedState savedState;
    try {
        savedState = ResourcesPlugin
            .getWorkspace()
            .addSaveParticipant (
                FavoritesActivator.getDefault(),
```



```
        saveParticipant());
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        // Recover if necessary.
        return;
    }

    if (savedState != null)
        savedState.processResourceChangeEvents(
            FavoritesManager.getManager());
}
```

提示 即使Eclipse是基于惰性插件初始化的，但它也提供了一种当工作台自身启动时就启动插件的机制。要在启动时激活，插件必须扩展org.eclipse.ui.startup扩展点并实现org.eclipse.ui.IStartup接口。一旦启动了插件，工作台将调用该插件的earlyStartup()方法（参见3.4.2节）。一个工作台首选项提供用户阻止插件早期启动的方法，因此请确保如果你的插件使用了该扩展点，它将在它没有早期启动的事件中合适地降级。

9.6 总结

本章论证了如何处理由系统产生的资源更改事件。当资源在任何时候被添加、修改或移除时，就生成了一个对应的更改事件。响应这些事件为你的插件提供了一种与Eclipse环境保持同步的方法。

参考文献

本书资源 (2.9节).

Arthorne, John, "How You've Changed! Responding to Resource Changes in the Eclipse Workspace," OTI, November 23, 2004 (www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html).



第10章 透 视 图

透视图 (Perspective) 是一种对Eclipse视图和命令进行分组以用于特殊任务 (如编写代码或调试) 的方法。包括多个插件的更大的Eclipse改进项可能提供它们自己的透视图。它仅包括一两个插件, 并且只提供一两个新Eclipse视图的小一些的Eclipse改进项一般改进现有透视图, 而不是提供全新的透视图。

本章将通过创建一个新的透视图进一步扩展Favorites示例。该透视图包含了Favorites视图并显示如何将Favorites视图添加至已有的透视图。

10.1 创建透视图

为了创建一个新的透视图, 扩展org.eclipse.ui.perspectives扩展点, 然后通过创建一个实现IPerspectiveFactory接口的透视图工厂类定义透视图的布局 (图10-1)。

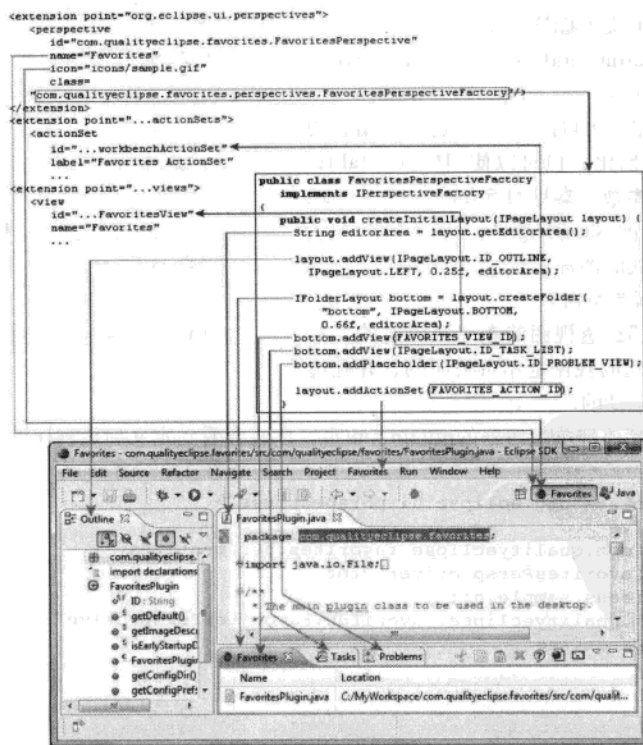


图10-1 透视图声明与行为

10.1.1 透视图扩展点

我们从打开Favorites插件清单编辑器开始。然后选择Extensions选项卡，并点击Add按钮。当打开New Extension向导时，从所有可用扩展点列表中选择org.eclipse.ui.perspectives（图10-2）。点击Finish按钮以将该扩展项添加至插件清单。

添加该扩展项将立即向插件清单的Extensions页添加一个新的透视图，名为com.qualityeclipse.favorites.perspective1。请注意，你可以右键点击org.eclipse.ui.perspectives扩展项并选择New > perspective以在任何时候添加附加的透视图。点击该新的透视图将在编辑器的右边显示它的属性（图10-3）。根据以下内容对它们进行修改：

- id——“com.qualityeclipse.favorites.FavoritesPerspective”

用于引用透视图的唯一标识符。

- name——“Favorites”

与透视图关联的文本标签。

- class——“com.qualityeclipse.favorites.perspectives.FavoritesPerspectiveFactory”

该类描述了透视图的布局。该类通过使用它的无参数构造函数初始化，但可以使用IExecutable-Extension接口赋予参数（参见21.5节）。

- icon——“icons/sample.gif”

与该透视图关联的图标。

- fixed——（留为空白）

如果为true，那么透视图的布局是固定的，并且由透视图工厂创建的视图是不可关闭的。并且透视图的布局是不可移动的。

如果你切换至插件清单编辑器的plugin.xml页，你将会看到定义新透视图的XML文档的以下新片段：

```
<extension point="org.eclipse.ui.perspectives">
  <perspective
    class="com.qualityeclipse.favorites.perspectives.
      FavoritesPerspectiveFactory"
    icon="icons/sample.gif"
    id="com.qualityeclipse.favorites.FavoritesPerspective"/>
  name="Favorites"
</extension>
```

10.1.2 透视图工厂

当指定透视图工厂类的名称时，点击class字段旁的Browse...按钮将打开一个类选择编辑器。在

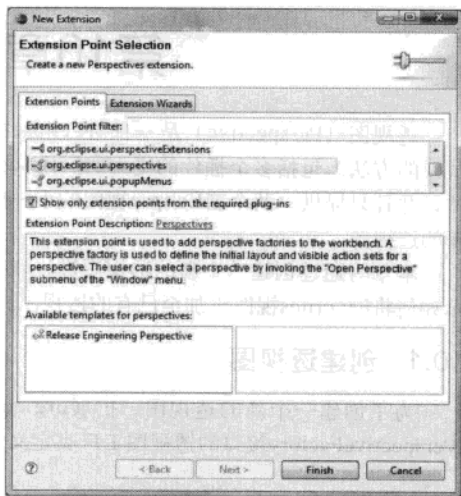


图10-2 显示选中org.eclipse.ui.perspectives扩展点的新建扩展项向导

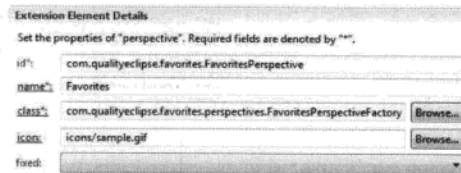


图10-3 收藏夹透视图的扩展元素的细节

该编辑器中可以选择一个已有的类。点击class字段右侧的class:标签将打开一个New Java Class向导。在该向导中可以创建新的类（遵循IPerspectiveFactory接口）（图10-4）。

IPerspectiveFactory接口定义了一个单独的方法，createInitialLayout()。它指定了初始页面的布局和透视图的可见操作集。工厂仅用于定义透视图的初始布局，并且然后就被忽略。默认地，布局区域包含了编辑器的空间，而不是视图的。工厂可以添加附加视图。这些视图被放置于相对于编辑器区域或另一个视图的位置。

打开新创建的FavoritesPerspectiveFactory类，并根据以下内容修改它。这样，Favorites视图将出现于编辑器区域的下方，并且标准Outline视图将会在它的左侧显示。

```
package com.qualityeclipse.favorites.perspectives;

import org.eclipse.ui.*;

public class FavoritesPerspectiveFactory
    implements IPerspectiveFactory
{
    private static final String FAVORITES_VIEW_ID =
        "com.qualityeclipse.favorites.views.FavoritesView";
    private static final String FAVORITES_ACTION_ID =
        "com.qualityeclipse.favorites.workbenchActionSet";

    public void createInitialLayout(IPageLayout layout) {
        // Get the editor area.
        String editorArea = layout.getEditorArea();

        // Put the Outline view on the left.
        layout.addView(
            IPageLayout.ID_OUTLINE,
            IPageLayout.LEFT,
            0.25f,
            editorArea);

        // Put the Favorites view on the bottom with
        // the Tasks view.
        IFolderLayout bottom =
            layout.createFolder(
                "bottom",
                IPageLayout.BOTTOM,
                0.66f,
                editorArea);
        bottom.addView(FAVORITES_VIEW_ID);
        bottom.addView(IPageLayout.ID_TASK_LIST);
        bottom.addPlaceholder(IPageLayout.ID_PROBLEM_VIEW);

        // Add the Favorites action set.
    }
}
```

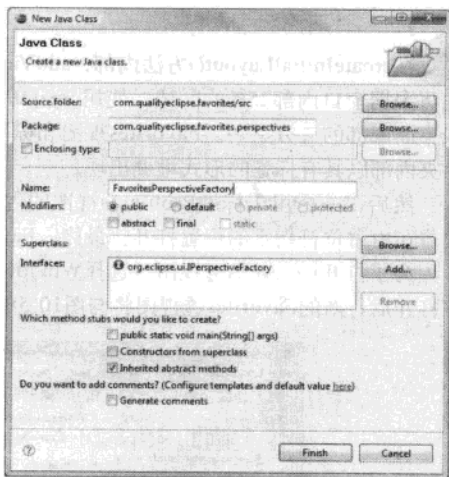


图10-4 Java类选择向导

```

        layout.addActionSet(FAVORITES_ACTION_ID);
    }
}

```

在createInitialLayout()方法内部，addView()方法用于添加标准Outline视图至编辑器区域的左侧。它将占据窗口内部25%的区域。使用createFolder()方法，可以创建一个文件夹布局以占据编辑器区域下面底部的三分之一。Favorites视图和标准Tasks视图被添加至文件夹布局，使得每一个都将在文件夹内部以具有标签的形式堆叠出现。

然后，一个用于标准Problems视图的占位符被添加至该文件夹。如果用户打开Problems视图，它将在由占位符指定的位置打开。最终，Favorites操作集被设为在透视图内部是默认可见的。

为了打开Favorites透视图，选择Window > Open Perspective > Other...，然后选择Favorites。当被打开后，新的Favorites透视图将与图10-5中显示的类似。

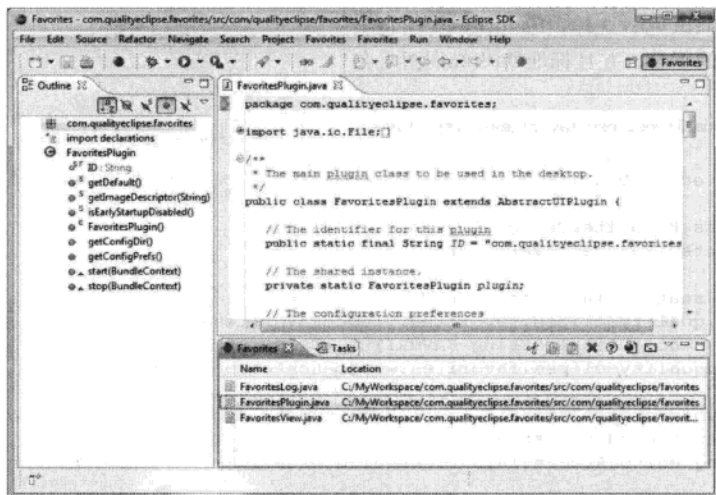


图10-5 收藏夹透视图

10.1.3 IPageLayout

如同在上一节所见的那样，IPageLayout接口定义了支持关于任意可能的透视图布局的必需的协议。它定义了许多常用API，包括有：

- addActionSet(String)——将具有给定ID的操作集添加至页面布局。
- addFastView(String)——将具有给定ID的视图添加至页面布局以作为一个快速视图。
- addFastView(String, float)——将具有给定ID的视图添加至页面布局以作为一个具有给定宽度比例的快速视图。
- addNewWizardShortcut(String)——添加一个创建向导至File New菜单。
- addPerspectiveShortcut(String)——添加一个透视图快捷方式至Perspective菜单。
- addPlaceholder(String, int, float, String)——为具有给定ID的视图添加一个占位符至页面布局。
- addShowInPart(String)——添加一个项至Show In提示框。

- addShowViewShortcut(String)——添加一个视图至Show View菜单。
- addStandaloneView(String, boolean, int, float, String)——添加一个具有给定ID的独立视图至该页面布局。独立视图不能被其他视图停靠。
- addView(String, int, float, String)——添加一个具有给定ID的视图至页面布局。
- createFolder(String, int, float, String)——创建并添加一个具有给定ID的文件夹至页面布局。
- createPlaceholderFolder(String, int, float, String)——为一个具有给定ID的新文件夹创建并添加一个占位符至页面布局。
- getEditorArea()——返回页面布局中的编辑器区域的特定ID。
- getViewLayout(String)——返回该页面布局中具有给定的复合ID的视图布局或占位符。
- setEditorAreaVisible(boolean)——显示或隐藏页面布局的编辑器区域。
- setFixed(boolean)——设置该布局是否是固定的。在固定布局中，布局组件不能被移动或改变大小，并且视图的初始集不能被关闭。

10.2 改进已有透视图

除了创建一个新视图之外，你也可以扩展一个已有的视图。可以通过添加新视图、占位符、快捷方式和操作集来扩展已有的视图。为了说明这一点，你可以添加几个扩展项至标准Resource透视图。

为了扩展已有的透视图，打开Favorites插件清单编辑器，选择Extensions选项卡，并点击Add按钮以打开New Extension向导。从可用扩展点列表选择org.eclipse.ui.perspectiveExtensions扩展点（图10-6）。

添加扩展项立即添加了一个名为com.quality.eclipse.favorites.perspectiveExtension1的新的透视图扩展项至插件清单的Extensions页面。请注意，你可以右键点击org.eclipse.ui.perspectiveExtensions扩展项并选择New >perspective Extension以显示它的属性并更改targetID字段为“org.eclipse.ui.resourcePerspective”（图10-7）。如同Extensions页面中看到的那样，这将更改透视图扩展项的名称。

提示 如果你想要透视图扩展项出现在所有透视图图中，使用“*”作为targetID。

当已经创建好了透视图扩展项时，可以添加一些不同的扩展项类型。这些扩展项类型包括视图、占位符和操作集，以及视图快捷方式、透视图和新建向导。

10.2.1 添加视图和占位符

视图可以直接被添加至已有透视图或可以被添加的占位符。这样，当用户打开视图时，它将出

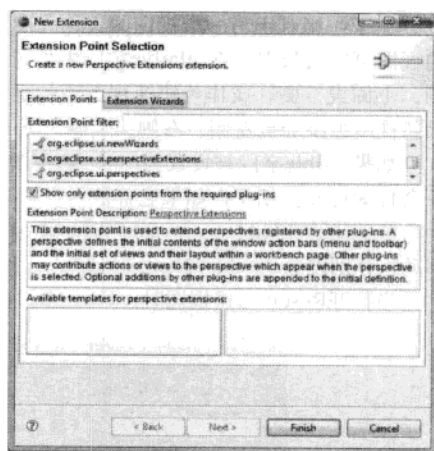


图10-6 显示选中org.eclipse.ui.perspectiveExtensions扩展点的新建扩展项向导

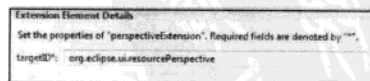


图10-7 扩展元素细节显示透视图扩展项的属性

现于正确的位置。作为示例，添加Favorites视图至标准Resource透视图。

在Extensions页面，点击新创建的org.eclipse.ui.resourcePerspective扩展项并选择New > view。这将立即添加一个名为com.quality-eclipse.favorites.view1的透视图视图扩展项至插件清单。点击该新的扩展项将显示它的属性。根据以下内容对这些属性作出更改（图10-8）。

- id——“com.qualityeclipse.favorites.views.FavoritesView”

Favorites视图的唯一标识符。

- relationship——“stack”

这项内容指定了视图应如何指向相对于目标视图的位置。

- relative——“org.eclipse.ui.views.TaskList”

被添加的视图应被指向并相对于的视图。

- visible——“true”

视图应在初始时可见。

如同在Extensions页面看到的那样，透视图视图扩展项的名称将会改变以反映输入 id。

除了在相对于另一个视图的文件夹中堆叠之外，被添加的视图可以被放置于在relative字段中指定的视图的左侧、右侧、上侧或下侧，或作为快速视图添加至左侧的工具栏。如果新视图被添加至左侧、右侧、上侧或下侧，还可以指定新视图从旧视图获取的空间的比例。

如果visible字段被设置为true，新视图将会在打开透视图时打开。如果它被设置为false，视图将不会自动打开。相反，如果它曾被用户打开过，则将会创建一个定义视图初始位置的占位符。

其他比如closable、movable和standalone的属性影响用户可以如何操作视图。要防止视图被关闭或移动，分别设置closable或movable为false。设置standalone属性为true将防止视图与其他任何视图堆叠。

当打开Resource透视图时，Favorites视图将会堆叠出现于相对于Tasks视图的位置（图10-9）。

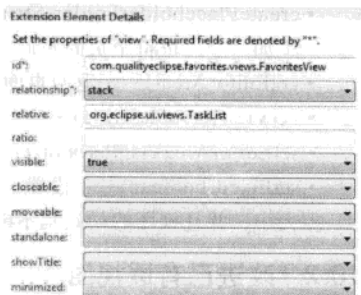


图10-8 扩展元素细节显示透视图视图扩展项的属性

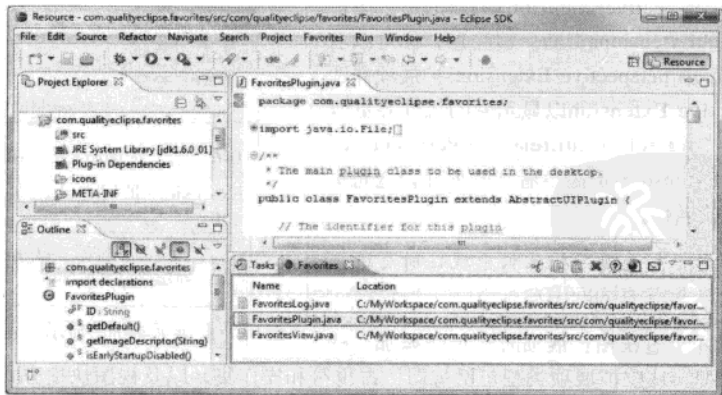


图10-9 显示收藏夹视图的资源透视图

提示 在声明新的透视图扩展项后，你必须为该新的透视图扩展项选择Window > Reset Perspective...，以使它出现在一个已打开的透视图 中。

切换到插件清单编辑器的plugin.xml页面，你将看到定义新透视图扩展项的XML文档的以下新的部分。

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.eclipse.ui.resourcePerspective">
    <view
      id="com.qualityeclipse.favorites.views.FavoritesView"
      minimized="false"
      relationship="stack"
      relative="org.eclipse.ui.views.TaskList"
      visible="true">
    </view>
  </perspectiveExtension>
</extension>
```

10.2.2 添加快捷方式

用于快速访问相关视图、透视图和新建向导的快捷方式也可以被添加至透视图。作为示例，添加用于访问Favorites视图和透视图的快捷方式至Resources透视图。

我们从添加一个用于从Resource透视图访问Favorites视图的视图快捷方式开始。在Extensions页，右键点击org.eclipse.ui.resourcePerspective扩展项并选择New > view Shortcut。这将添加一个名为com.qualityeclipse.favorites.viewShortcut1的视图快捷方式至插件清单。点击该快捷方式以显示它的属性，然后将id字段更改为“com.qualityeclipse.favorites.views.FavoritesView”（图10-10）。这将更改在Extensions页面看到的视图快捷方式扩展项的名称。

当打开资源透视图时，这将在Window > Show View菜单添加一个快捷方式至Favorites视图（图10-11）。

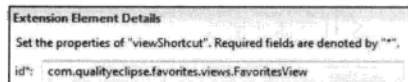


图10-10 显示视图快捷方式扩展项属性的扩展项元素细节

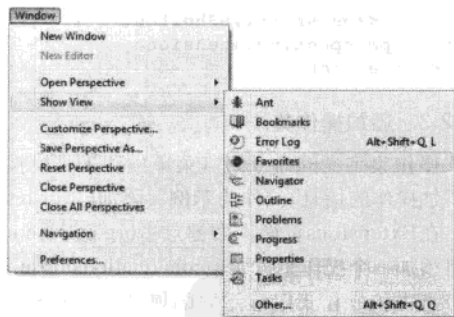


图10-11 显示收藏夹快捷方式的Show View菜单

然后，添加一个用于从资源透视图访问Favorites透视图的透视图快捷方式。在Extensions页，右键点击org.eclipse.ui.resourcePerspective扩展项并选择New > perspectiveShortcut。这将添加一个名为com.qualityeclipse.favorites.perspectiveShortcut1的透视图快捷方式扩展项至插件清单。点击它以显示它的属性，然后将id字段更改为“com.qualityeclipse.favorites.FavoritesPerspective”（图10-12）。这将更改在Extensions页中看到的透视图快捷方式扩展项的名称。

当打开Resource透视图时，这将在Window > Open Perspective菜单添加一个快捷方式至

Favorites视图 (图10-13)。

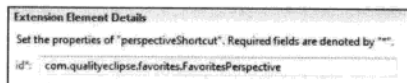


图10-12 显示透视图快捷方式扩展
项属性的扩展元素细节

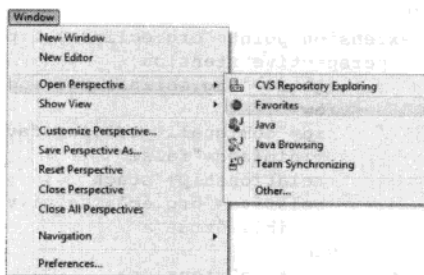


图10-13 显示收藏夹快捷方式的打开透视图菜单

如果你切换到插件清单编辑器的plugin.xml页面, 你将会看到定义新视图和透视图快捷方式的XML文档的以下部分:

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.eclipse.ui.resourcePerspective">
    ...
    <viewShortcut
      id="com.qualityeclipse.favorites.views.FavoritesView">
    </viewShortcut>
    <perspectiveShortcut
      id="com.qualityeclipse.favorites.FavoritesPerspective">
    </perspectiveShortcut>
    </perspectiveExtension>
</extension>
```

10.2.3 添加操作集

操作集定义的命令组 (菜单项和工具栏按钮) 也可以添加至透视图 (参见第6章以了解更多关于添加操作的信息)。作为示例, 添加Favorites操作集至Resources透视图。

在Extensions页面, 右键点击org.eclipse.ui.resourcePerspective扩展项并选择New > actionSet。这将添加一个操作集扩展项com.qualityeclipse.favorites.actionSet1至插件清单。

点击该操作集以显示它的属性, 并修改该id字段为“com.qualityeclipse.favorites.workbenchActionSet” (图10-14)。这将更改Extensions页面中所看到的操作集扩展项的名称。

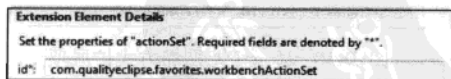


图10-14 显示操作集扩展项属性的扩展元素细节

如果你切换到插件清单编辑器的plugin.xml页面, 你将会看到定义新操作集扩展项的XML文档的以下被添加片段。

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.eclipse.ui.resourcePerspective">
    ...
    <actionSet
      id="com.qualityeclipse.favorites.workbenchActionSet">
```

```
</actionSet>
</perspectiveExtension>
</extension>
```

当完成以上改进后,新的透视图和透视图扩展项在Extensions页是可见的(图10-15)。

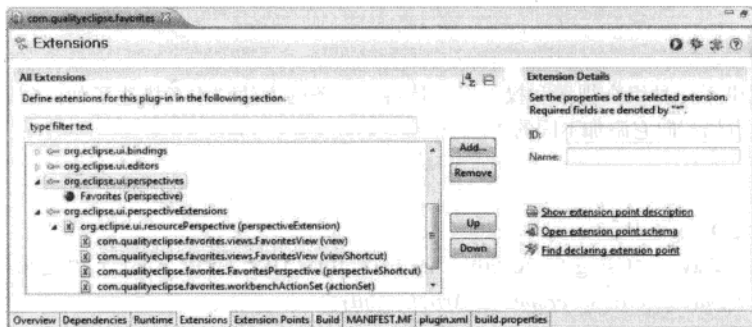


图10-15 显示新的透视图和透视图扩展项的扩展项页

10.3 RFRS相关事项

《RFRS Requirements》的“用户界面”一节包含了与透视图相关的三个最佳做法。它们都来源于Eclipse UI准则。

10.3.1 为长时间任务创建 (RFRS 5.3.5.10)

用户界面准则#8.1是一个最佳做法。它说明:

为包含更小的、非模态的任务执行的长时间任务创建一个新的透视图类型。

新透视图类型应在存在一组相关任务时创建。这一组任务将从操作和视图的预定义配置中获益,并且这些任务是长时间的。一个面向任务的方法是必要的。

为了通过这项测试,创建一个由你的程序定义的透视图的列表,并说明它们的用途。对于每一个透视图,描述视图、快捷方式、新建向导项和包含的操作集。在本章稍早展示的示例的情况中,展示Favorites透视图(图10-5)并向审阅者描述它的用途。

10.3.2 扩展已有透视图 (RFRS 5.3.5.11)

用户界面准则#8.2是一个最佳做法。它说明:

为了显示单独的一个或两个视图,扩展一个已有的透视图类型。

如果插件仅添加少量视图,并且它们都是扩展一个已有的任务,最好添加这些视图至一个已有的透视图。比如,如果你创建一个扩展Java代码创建任务的视图,不要创建新的透视图。作为替代,添加它至已有的Java透视图。这种方法提供了与现有平台更好的集成。

对于这项测试,简单地创建一个所有你的程序添加至其他已有透视图的视图的列表。在之前的示例中,展示被添加至Resource透视图的Favorites视图(图10-9)。

10.3.3 添加操作至窗口菜单 (RFRS 5.3.5.15)

用户界面准则#8.6是一个最佳做法。它说明:

向窗口菜单栏添加适用于透视图的任务导向的操作和操作集，或适用于任何更大的工作流的操作和操作集。

对于这项测试，提供一个包含已经被添加至所有透视图的操作集的列表。对于Favorites视图来说，列出与Favorites操作集类似的项。该操作集在10.2.3节被添加至Resource透视图。

10.4 总结

透视图提供了一种组合视图和操作以支持特定任务的方法。本章描述了如何创建新的透视图，定义它的默认布局和向它添加不同的扩展项。

参考文献

本书资源(2.9节).

Springgay, Dave, "Using Perspectives in the Eclipse UI," OTI, August 27, 2001 (www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html).

Lorimer, R. J., "Eclipse: Create Your Own Perspective," April 8, 2005 (www.javalobby.org/java/forums/t18187.html).



第11章 对话框与向导

好的UI准则建议开发者创建非模态的Eclipse编辑器和视图，但有时候模态对话框或向导更合适。本章展开讨论Eclipse的对话框和向导的框架，讨论何时对话框或向导应被用于替代视图或编辑器；提供不同的示例，并讨论Eclipse的不同的内建对话框类。

11.1 对话框

无论何时在非模态样式中用户请求信息或向用户展示信息时，都应允许用户自由地与工作台中的所有资源进行交互。窗口、页面、编辑器和视图全都是非模态UI组件。这些组件没有限制用户与它们交互的顺序。对话框一般都是模态的。它限制用户输入要求的信息或取消操作。应使用模态UI组件的唯一时机是，在继续任何其他操作之前程序限制需要收集或分发信息。即使是这样，时间也应当尽可能短。

在一种情况下，程序可以使用对话框展示文件的两种不同版本。不幸的是，这种方法阻止用户在比较间和一些其他UI组件（如另一个编辑器或视图）间切换。一种更好的方法应是在一个比较编辑器中展示相同的信息。

创建新项目展示了一种不同的情形。在这种情况下，在可以执行前，操作必须顺序收集所有的必需信息。用户已经请求操作并在收集了所有信息，在操作完成之前，一般不需要与程序的另一部分交互。在这种情况下，对话框或向导是更为合理的。

11.1.1 SWT对话框与JFace对话框

在Eclipse中存在两种截然不同的对话框层次结构。SWT对话框（org.eclipse.swt.Dialog）是平台内建对话框的Java表示，如文件对话框或字体对话框。正因为如此，它们是不可移动或扩展的。JFace对话框（org.eclipse.jface.dialogs.Dialog）是独立于平台的对话框。基于它可以创建向导。这里，仅简单讨论SWT对话框，而JFace对话框将得到详细的说明。

11.1.2 普通SWT对话框

Eclipse包含了几个SWT对话框类。这些类提供了用于底层平台特定的对话框的独立于平台的接口：

- ColorDialog——提示用户从一个可用颜色预定义集中选择一种颜色。
- DirectoryDialog——提示用户浏览文件系统并选择一个目录。合法的样式包括SWT.OPEN用于选择一个已有的目录，SWT.SAVE用于指定一个新目录。
- FileDialog——提示用户浏览文件系统并选择或输入一个文件名。合法的样式包括SWT.OPEN用于选择一个已有的文件，SWT.SAVE用于指定一个新文件。
- FontDialog——提示用户从所有可用字体中选择一种字体。
- MessageBox——向用户显示一条信息。合法的图标样式在表11-1中列出。

合法的按钮样式包括：

SWT.OK

```

SWT.OK | SWT.CANCEL
SWT.YES | SWT.NO
SWT.YES | SWT.NO | SWT.CANCEL
SWT.RETRY | SWT.CANCEL
SWT.ABORT | SWT.RETRY | SWT.IGNORE

```






• **PrintDialog**——提示用户在开始一项打印任务之前选择打印机和不同的打印相关参数。

使用这些SWT对话框，可以指定以下模态样式中的

一种：

- **SWT.MODELESS**——非模态对话框行为。
- **SWT.PRIMARY_MODAL**——考虑了父shell的模态行为。
- **SWT.APPLICATION_MODAL**——考虑了程序的模态行为。
- **SWT.SYSTEM_MODAL**——考虑了整个系统的模态行为。

表11-1 图标样式

常 量	图 标
SWT.ICON_ERROR	
SWT.ICON_INFORMATION	
SWT.ICON_QUESTION	
SWT.ICON_WARNING	
SWT.ICON_WORKING	

11.1.3 普通JFace对话框

Eclipse中存在多种JFace对话框。它们既可以被直接初始化，又可以通过继承重用。

抽象对话框

- **AbstractElementListSelectionDialog**——用于从元素列表中选择元素的抽象对话框。
- **IconAndMessageDialog**——具有一个图标和一个消息作为头两个小部件的对话框的抽象超类。
- **SelectionDialog**——显示并返回一个选择的抽象对话框。
- **SelectionStatusDialog**——具有一个状态栏和OK/Cancel按钮的对话框的抽象基础类。状态信息必须作为一个StatusInfo对象进行传递，并且可以是一个错误、警告或确认信息。OK按钮根据状态被设置为可用或不可用。
- **StatusDialog**——具有一个状态栏和OK/Cancel按钮的对话框的抽象基础类。
- **TitleAreaDialog**——抽象对话框。该对话框具有一个标题区域用于显示标题和图像，和一个通用区域用于显示描述、消息或错误信息。
- **TrayDialog**——可以在边缘包含一个任务栏的特殊对话框。

文件对话框

- **SaveAsDialog**——向用户请求路径的标准“另存为”对话框。**getResult()**方法返回路径。请注意给定路径中的文件夹可能不存在且可能需要被创建。

信息对话框

- **ErrorDialog**——向用户显示一个或多个错误的对话框。这些错误包含在一个IStatus对象中。如果错误包含附加的信息细节，那么将自动添加一个Details按钮。该按钮在被用户按下时显示或隐藏错误细节查看器（参见11.1.9节以了解符合RFRS要求的类似对话框）。
- **MessageDialog**——向用户显示消息的对话框。
- **MessageDialogWithToggle**——允许用户调整切换设置的MessageDialog。如果提供了参数存储并且用户选择切换，那么用户的回答（是/好或不是）将会保存在存储中（参见12.3.2节）。如果没有提供存储位置，那么该信息可以在对话框关闭后查询。

资源对话框

- `ContainerSelectionDialog`——向用户请求容器资源的标准选择对话框。`getResult()`方法返回被选中的容器资源。
- `NewFolderDialog`——用于创建新文件夹的对话框。可以选择文件夹是否被链接至文件系统文件夹。
- `ProjectLocationMoveDialog`——用于选择要移动的项目的位置的对话框。
- `ProjectLocationSelectionDialog`——用于选择要复制的项目的名称和位置的对话框。
- `ResourceListSelectionDialog`——向用户显示资源列表。该列表包含一个字符串样式的文本条目字段。该字符串样式用于过滤资源列表。
- `ResourceSelectionDialog`——向用户请求资源列表的标准资源选择对话框。`getResult()`方法返回被选中的资源。
- `TypeFilteringDialog`——允许用户选择文件编辑器的选择对话框。

选择对话框

- `CheckedTreeSelectionDialog`——从树形结构中选择元素的对话框。
- `ContainerCheckedTreeViewer`——容器（非叶）节点具有特殊选中/灰色状态的增强的`CheckedTreeSelectionDialog`对话框。
- `ElementListSelectionDialog`——从元素列表中选择元素的对话框。
- `ElementTreeSelectionDialog`——从树形结构中选择元素的对话框。
- `ListDialog`——提示从元素列表选择一个元素的对话框。使用`IStructuredContentProvider`以提供元素，使用`ILabelProvider`以提供它们的标签。
- `ListSelectionDialog`——向用户请求选择列表的标准对话框。该类使用一个由内容和标签提供者对象表示的抽象数据模型进行配置。`getResult()`方法返回被选中元素。
- `TwoPaneElementSelector`——具有两个窗格的列表选择对话框。重复的条目将会被折叠至一起并在较低的窗格（合格器）中显示。

其他对话框

- `InputDialog`——用于向用户请求输入字符串的简单输入对话框。
- `MarkerResolutionSelectionDialog`——允许用户从标记决定列表中选择相关项的对话框。
- `ProgressMonitorDialog`——在长时间运行的操作时期内显示进度的模态对话框（参见9.4节）。
- `TaskPropertiesDialog`——显示一个新建或已有的任务，或问题的属性。
- `WizardDialog`——显示一个向导并实现`IWizard` `Container`接口（参见11.2.3节）的对话框。

11.1.4 创建JFace对话框

`Dialog`类的默认实现创建了一个包含内容区域的对话框。该内容区域包含对话框特定的控件，在其下部还具有一个包含OK和Cancel按钮的按钮栏（图11-1）。

一般地，新的对话框通过继承`org.eclipse.jface.dialogs.Dialog`创建，并通过覆盖一些方法为特定用途自定义对话框。

- `buttonPressed(int)`——当用户点击由`createButton`方法创建的按钮时调用该方法。如果OK按钮被

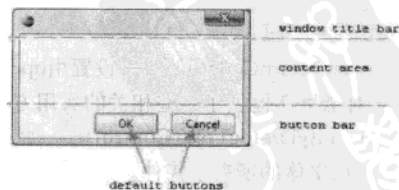


图11-1 默认对话框结构

按下，默认实现将调用okPressed()；如果Cancel按钮被按下，默认实现将调用cancelPressed()。

- cancelPressed()——当用户按下Cancel按钮时调用。默认实现设置返回代码为Window.CANCEL并关闭对话框。
- close()——关闭对话框，释放它的shell，并从它的窗口管理器（如果有一个）移除对话框。
- createButton(Composite, int, String, boolean)——使用给定的标识符和标签在按钮栏中创建并返回一个新按钮。该方法一般从createButtonsForButtonBar方法调用。
- createButtonBar(Composite)——放置一个按钮栏并调用createButtonsForButtonBar方法以操作它。子类在需要时可以覆盖createButtonBar或createButtonsForButtonBar。
- createButtonsForButtonBar(Composite)——在按钮栏中创建按钮。默认实现在右下角创建OK和Cancel按钮。子类可以覆盖该方法以替代默认按钮，或使用createButton方法扩展该方法以扩展默认按钮。
- createContents(Composite)——为按钮栏上面的对话框创建并返回内容区域。子类一般调用超类的方法，然后添加控件至返回的复合组件。
- createDialogArea(Composite)——为按钮栏上面的对话框创建并返回内容区域。子类一般调用超类方法，然后添加控件至返回的复合组件。
- okPressed()——当用户按下OK按钮时调用。默认实现设置返回代码为Window.OK并关闭对话框。
- open()——打开该对话框。如果它还没有被创建就先创建它。该方法将等待直至用户关闭对话框，然后返回对话框的返回代码。对话框的返回代码是对话框相关的，预定义了两个标准返回代码：Window.OK和Window.CANCEL。
- setShellStyle(int)——设置shell样式位以用于创建对话框。该方法在shell被创建后不起作用。

合法的样式位包括：

```
SWT.MODELESS  
SWT.PRIMARY_MODAL  
SWT.APPLICATION_MODAL  
SWT.SYSTEM_MODAL  
SWT.SHELL_TRIM  
SWT.DIALOG_TRIM  
SWT.BORDER  
SWT.CLOSE  
SWT.MAX  
SWT.MIN  
SWT.RESIZE  
SWT.TITLE
```

- setReturnCode(int)——设置由open()方法返回的对话框的返回代码。

对话框还提供了一些相关的实用方法：

- applyDialogFont(Control)——将对话框字体应用于指定的控件，并递归地应用于当前具有默认字体的所有子控件。
- getImage(String)——返回具有给定键值的标准对话框图像（Dialog.DLG_IMG_*常量之一）。这些图像由对话框框架管理并且不能由另一个组关闭。
- shortenText(String, Control)——通过在必要时插入英文省略号（“...”）缩短指定文本，以使它用像素表示的宽度不会超过给定控件的宽度。

11.1.5 对话框单元

如果你在对话框区域基于绝对定位（null布局）放置控件而不使用布局管理器，如GridLayout或FormLayout，那么当使用不同字体时将出现问题。如果对话框是基于一种字体并使用一种像素大小设置大小，而用户具有另一种像素大小的字体的系统集，那么控件对于使用的字体来说将会太大或太小。要减少该问题的发生几率，你需要基于字体的平均字符大小或对话框单元来放置控件并设置其大小（图11-2）。



图11-2 对话框单元根据字母“T”叠加

对话框单元基于当前字体并独立于显示设备；因此，它们可以用于在对话框内放置控件，并独立于当前使用的字体。它们被定义为一个字符的平均宽度的四分之一和一个字符的平均高度的八分之一。

```
dialog unit X = average character width / 4
dialog unit Y = average character height / 8
```

因此，使用以下代码从对话框单元转换至像素。

```
pixelX = (dialog unit X * average character width) / 4
pixelY = (dialog unit Y * average character height) / 8
```

Eclipse对话框框架提供了几个便利的方法用于转换对话框单元或字符大小至像素大小。

- `convertHeightInCharsToPixels(int)`——返回对应于给定数量字符的高度的像素的数量。
- `convertHorizontalDLUsToPixels(int)`——返回对应于给定数量的水平对话框单元的像素的数量。
- `convertVerticalDLUsToPixels(int)`——返回对应于给定数量的竖直对话框单元的像素的数量。
- `convertWidthInCharsToPixels(int)`——返回对应于给定数量字符的宽度的像素的数量。

11.1.6 对话框的初始位置和大小

由对话框框架实现的对话框的默认行为用于在由对话框的构造函数指定的父窗口的最上层初始放置对话框。要为对话框提供不同的初始位置或大小，你应在需要时覆盖以下方法。

- `getInitialLocation(Point)`——返回对话框使用的初始位置。默认实现将对话框相对于父shell或如果没有父shell时相对于显示边界，水平（距离左右边界相等的中间位置）和竖直（距离上边界三分之一和下边界三分之二）放置对话框。该参数是对话框的初始位置。它由 `getInitialSize()` 方法返回。
- `getInitialSize()`——返回对话框使用的初始大小。默认实现基于使用 `computeSize` 方法的对话框的布局 and 控件返回对话框的首选大小。

11.1.7 可调整大小的对话框

默认情况下，Dialog的子类是不可调整大小的。但在Eclipse 3.4中你可以覆盖 `isResizable` 方法来使得对话框可以调整大小：

```
protected boolean isResizable() {
    return true;
}
```

为了保存跨引用使用的对话框的大小和位置，你必须提供一个位置用于存储值。一旦你实现了

isResizable和getDialogBoundsSettings, 用户就可以重新调整对话框的大小, 并且该大小将在多个实例之间得以保存:

```
private static final String RESIZABLE_DIALOG_SETTINGS =
    "MyResizableDialogSettings";

protected IDialogSettings getDialogBoundsSettings() {
    IDialogSettings settings =
        FavoritesActivator.getDefault().getDialogSettings();
    IDialogSettings section =
        settings.getSection(RESIZABLE_DIALOG_SETTINGS);
    if (section == null)
        section = settings.addNewSection(RESIZABLE_DIALOG_SETTINGS);
    return section;
}
```

为了了解更多关于IDialogSettings的内容, 参见11.2.7节。

11.1.8 收藏夹视图过滤器对话框

作为示例, 为收藏夹视图创建一个特殊的过滤器对话框。该对话框向用户展示了选择基于名称、类型或位置的过滤内容 (参见7.2.7节和7.3.4节)。该对话框限制它自身以向用户展示信息并从用户收集信息, 以及为过滤器操作提供访问方法。我们从创建一个新的FavoritesFilterDialog类开始:

```
public class FavoritesFilterDialog extends Dialog
{
    private String namePattern;
    private String locationPattern;
    private Collection<FavoriteItemType> selectedTypes;
    public FavoritesFilterDialog(
        Shell parentShell,
        String namePattern,
        String locationPattern,
        FavoriteItemType[] selectedTypes
    ) {
        super(parentShell);
        this.namePattern = namePattern;
        this.locationPattern = locationPattern;
        this.selectedTypes = new HashSet<FavoriteItemType>();
        for (int i = 0; i < selectedTypes.length; i++)
            this.selectedTypes.add(selectedTypes[i]);
    }
}
```

然后, 覆盖createDialogArea()方法以创建出现于对话框的上面部分的不同字段。

```
private Text namePatternField;
private Text locationPatternField;

protected Control createDialogArea(Composite parent) {
    Composite container = (Composite) super.createDialogArea(parent);
    final GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 2;
    container.setLayout(gridLayout);

    final Label filterLabel = new Label(container, SWT.NONE);
    filterLabel.setLayoutData(new GridData(GridData.BEGINNING,
        GridData.CENTER, false, false, 2, 1));
    filterLabel.setText("Enter a filter (* = any number of "
```

```

        + "characters, ? = any single character)"
        + "\nor an empty string for no filtering:");

final Label nameLabel = new Label(container, SWT.NONE);
nameLabel.setLayoutData(new GridData(GridData.END,
    GridData.CENTER, false, false));
nameLabel.setText("Name:");

namePatternField = new Text(container, SWT.BORDER);
namePatternField.setLayoutData(new GridData(GridData.FILL,
    GridData.CENTER, true, false));

final Label locationLabel = new Label(container, SWT.NONE);
final GridData gridData = new GridData(GridData.END,
    GridData.CENTER, false, false);
gridData.horizontalIndent = 20;
locationLabel.setLayoutData(gridData);
locationLabel.setText("Location:");

locationPatternField = new Text(container, SWT.BORDER);
locationPatternField.setLayoutData(new GridData(GridData.FILL,
    GridData.CENTER, true, false));
final Label typesLabel = new Label(container, SWT.NONE);
typesLabel.setLayoutData(new GridData(GridData.BEGINNING,
    GridData.CENTER, false, false, 2, 1));
typesLabel.setText("Select the types of favorites to be shown:");
final Composite typeCheckboxComposite = new Composite(container,
    SWT.NONE);
final GridData gridData_1 = new GridData(GridData.FILL,
    GridData.FILL, false, false, 2, 1);
gridData_1.horizontalIndent = 20;
typeCheckboxComposite.setLayoutData(gridData_1);
final GridLayout typeCheckboxLayout = new GridLayout();
typeCheckboxLayout.numColumns = 2;
typeCheckboxComposite.setLayout(typeCheckboxLayout);

return container;
}

```

然后，创建一个新的createTypeCheckboxes()方法。该方法在createDialogArea()方法的末尾被调用，为每一个类型创建一个单选框。

```

private Map typeFields;

protected Control createDialogArea(Composite parent) {
    ... existing code ...
    createTypeCheckboxes(typeCheckboxComposite);
    return container;
}

private void createTypeCheckboxes(Composite parent) {
    typeFields = new HashMap<FavoriteItemType, Button>();
    FavoriteItemType[] allTypes = FavoriteItemType.getTypes();
    for (int i = 0; i < allTypes.length; i++) {
        final FavoriteItemType eachType = allTypes[i];
        if (eachType == FavoriteItemType.UNKNOWN)
            continue;
        final Button button = new Button(parent, SWT.CHECK);

```

```

        button.setText(eachType.getName());
        typeFields.put(eachType, button);
        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent e) {
                if (button.getSelection())
                    selectedTypes.add(eachType);
                else
                    selectedTypes.remove(eachType);
            }
        });
    }
}

```

添加initContent()方法。该方法在createDialogArea()方法的末尾被调用以初始化对话框中的不同字段:

```

protected Control createDialogArea(Composite parent) {
    ... existing code ...
    createTypeCheckboxes(typeCheckboxComposite);
    initContent();
    return container;
}

private void initContent() {
    namePatternField.setText(namePattern != null ? namePattern : "");
    namePatternField.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            namePattern = namePatternField.getText();
        }
    });

    locationPatternField
        .setText(locationPattern != null ? locationPattern : "");
    locationPatternField.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            locationPattern = locationPatternField.getText();
        }
    });

    FavoriteItemType[] allTypes = FavoriteItemType.getTypes();
    for (int i = 0; i < allTypes.length; i++) {
        FavoriteItemType eachType = allTypes[i];
        if (eachType == FavoriteItemType.UNKNOWN)
            continue;
        Button button = typeFields.get(eachType);
        button.setSelection(selectedTypes.contains(eachType));
    }
}

```

覆盖configureShell()方法以设置对话框标题:

```

protected void configureShell(Shell newShell) {
    super.configureShell(newShell);
    newShell.setText("Favorites View Filter Options");
}

```

最后, 为客户端添加访问方法, 以在打开对话框时提取由用户指定的设置:

```

public String getNamePattern() {
    return namePattern;
}

```

```

    }

    public String getLocationPattern() {
        return locationPattern;
    }

    public FavoriteItemType[] getSelectedTypes() {
        return selectedTypes
            .toArray(new FavoriteItemType[selectedTypes.size()]);
    }
}

```

过滤器操作（参见7.3.4节中的FavoritesViewFilterAction）必须当用户使用OK按钮关闭对话框时被修改，使用当前过滤器设置填充对话框，打开对话框，和处理指定的过滤器设置。如果对话框是使用Cancel按钮或除了OK按钮之外的所有方法关闭的，根据标准对话框操作准则，更改将被放弃。在下列代码中引用的类型和位置查看过滤器留给读者作为练习。

```

public void run() {
    FavoritesFilterDialog dialog =
        new FavoritesFilterDialog(
            shell,
            nameFilter.getPattern(),
            typeFilter.getTypes(),
            locationFilter.getPattern());
    if (dialog.open() != InputDialog.OK)
        return;
    nameFilter.setPattern(dialog.getNamePattern());
    locationFilter.setPattern(dialog.getLocationPattern());
    typeFilter.setTypes(dialog.getSelectedTypes());
}

```

让上面的run()方法编译包括添加一个与已有的FavoritesViewNameFilter类似的新FavoritesViewLocationFilter和FavoritesViewTypeFilter。当完成这些更改后，过滤器对话框将在选中Filter...菜单项时向用户展示过滤器设置（图11-3）。

11.1.9 细节对话框

RFRS要求之一包括当向用户报告问题时，标识插件和插件创建者。也就是说，无论何时程序需要向用户报告错误消息或异常，插件的唯一标识符、版本和创建者必须在对话框中可见。org.eclipse.jface.dialogs.ErrorDialog可以通过设置ErrorSupportProvider显示该信息：

```

Policy.setErrorSupportProvider(new ErrorSupportProvider() {
    public Control createSupportArea(Composite parent, IStatus status) {
        ... create controls to display the provider and exception info ...
    }
}

```

遗憾的是，这是一个全局设置，因此它仅当你基于Eclipse RCP框架来创建你自己的程序时才适用。如果你改进已有程序的功能，那么你必须创建你自己的ExceptionDetailsDialog（图11-4）。它在细节部分显示异常和必要的产品信息。该细节部分根据RFRS标准使用一个Details按钮来显示或隐藏。

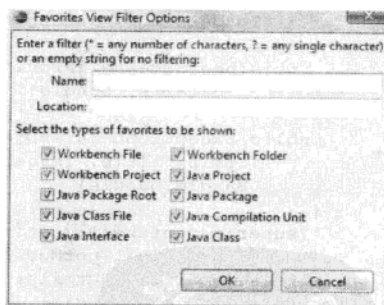


图11-3 新的收藏夹视图过滤器选项对话框

当按下Details按钮时，对话框调整它自身的大小以显示附加信息（图11-5）。

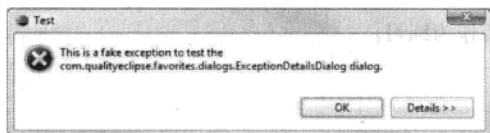


图11-4 隐藏细节的细节对话框



图11-5 显示细节的细节对话框

ExceptionDetailsDialog类实现了该扩展细节行为。

```
public class ExceptionDetailsDialog extends AbstractDetailsDialog {
    private final Object details;
    private final Bundle bundle;

    public ExceptionDetailsDialog(Shell parentShell, String title,
        Image image, String message, Object details, Bundle bundle)
    {
        this(new SameShellProvider(parentShell), title, image, message,
            details, bundle);
    }

    public ExceptionDetailsDialog(IShellProvider parentShell,
        String title, Image image, String message, Object details,
        Bundle bundle)
    {
        super(parentShell, getTitle(title, details), getImage(image,
            details), getMessage(message, details));
        this.details = details;
        this.bundle = bundle;
    }
}
```

有几个实用方法基于构造函数参数提供的信息创建内容。getTitle()方法返回基于提供的标题和细节对象的标题。

```
public static String getTitle(String title, Object details) {
    if (title != null)
        return title;
    if (details instanceof Throwable) {
        Throwable e = (Throwable) details;
        while ((e instanceof InvocationTargetException)
            e = ((InvocationTargetException) e).getTargetException());
        String name = e.getClass().getName();
        return name.substring(name.lastIndexOf('.') + 1);
    }
}
```

```

    }
    return "Exception";
}

```

getImage()方法返回基于提供的图像和细节对象的图像。

```

public static Image getImage(Image image, Object details) {
    if (image != null)
        return image;
    Display display = Display.getCurrent();
    if (details instanceof IStatus) {
        switch (((IStatus) details).getSeverity()) {
            case IStatus.ERROR :
                return display.getSystemImage(SWT.ICON_ERROR);
            case IStatus.WARNING :
                return display.getSystemImage(SWT.ICON_WARNING);
            case IStatus.INFO :
                return display.getSystemImage(SWT.ICON_INFORMATION);
            case IStatus.OK :
                return null;
        }
    }
    return display.getSystemImage(SWT.ICON_ERROR);
}

```

getMessage()方法和帮助方法创建基于提供的消息和细节对象的消息。

```

public static String getMessage(String message, Object details) {
    if (details instanceof Throwable) {
        Throwable e = (Throwable) details;
        while (e instanceof InvocationTargetException)
            e = ((InvocationTargetException) e).getTargetException();
        if (message == null)
            return e.toString();
        return MessageFormat.format(
            message, new Object[] { e.toString() });
    }
    if (details instanceof IStatus) {
        String statusMessage = ((IStatus) details).getMessage();
        if (message == null)
            return statusMessage;
        return MessageFormat.format(
            message, new Object[] { statusMessage });
    }
    if (message != null)
        return message;
    return "An Exception occurred.";
}

public static void appendException(PrintWriter writer, Throwable ex)
{
    if (ex instanceof CoreException) {
        appendStatus(writer, ((CoreException) ex).getStatus(), 0);
        writer.println();
    }
    appendStackTrace(writer, ex);
    if (ex instanceof InvocationTargetException)
        appendException(writer, ((InvocationTargetException) ex)

```

```

        .getTargetException());
    }
    public static void appendStatus(
        PrintWriter writer, IStatus status, int nesting
    ) {
        for (int i = 0; i < nesting; i++)
            writer.print(" ");
        writer.println(status.getMessage());
        IStatus[] children = status.getChildren();
        for (int i = 0; i < children.length; i++)
            appendStatus(writer, children[i], nesting + 1);
    }

    public static void appendStackTrace(
        PrintWriter writer, Throwable ex
    ) {
        ex.printStackTrace(writer);
    }
}

```

当点击Details按钮时，超类决定细节区域是否要显示或隐藏，以及在需要时，调用createDetailsArea()方法为细节区域创建内容。

```

protected Control createDetailsArea(Composite parent) {

    // Create the details area.
    Composite panel = new Composite(parent, SWT.NONE);
    panel.setLayoutData(new GridData(GridData.FILL_BOTH));
    GridLayout layout = new GridLayout();
    layout.marginHeight = 0;
    layout.marginWidth = 0;
    panel.setLayout(layout);

    // Create the details content.
    createProductInfoArea(panel);
    createDetailsViewer(panel);

    return panel;
}

protected Composite createProductInfoArea(Composite parent) {
    // If no bundle specified, then nothing to display here.
    if (bundle == null)
        return null;
    Composite composite = new Composite(parent, SWT.NULL);
    composite.setLayoutData(new GridData());
    GridLayout layout = new GridLayout();
    layout.numColumns = 2;
    layout.marginWidth = convertHorizontalDLUsToPixels(
        IDialogConstants.HORIZONTAL_MARGIN);
    composite.setLayout(layout);
    Dictionary<?, ?> bundleHeaders = bundle.getHeaders();
    String pluginId = bundle.getSymbolicName();
    String pluginVendor =
        (String) bundleHeaders.get("Bundle-Vendor");
    String pluginName = (String) bundleHeaders.get("Bundle-Name");
    String pluginVersion =
        (String) bundleHeaders.get("Bundle-Version");
}

```

```

    new Label(composite, SWT.NONE).setText("Provider:");
    new Label(composite, SWT.NONE).setText(pluginVendor);
    new Label(composite, SWT.NONE).setText("Plug-in Name:");
    new Label(composite, SWT.NONE).setText(pluginName);
    new Label(composite, SWT.NONE).setText("Plug-in ID:");
    new Label(composite, SWT.NONE).setText(pluginId);
    new Label(composite, SWT.NONE).setText("Version:");
    new Label(composite, SWT.NONE).setText(pluginVersion);

    return composite;
}

protected Control createDetailsViewer(Composite parent) {
    if (details == null)
        return null;
    Text text = new Text(parent, SWT.MULTI | SWT.READ_ONLY
        | SWT.BORDER | SWT.H_SCROLL | SWT.V_SCROLL);
    text.setLayoutData(new GridData(GridData.FILL_BOTH));

    // Create the content.
    StringWriter writer = new StringWriter(1000);
    if (details instanceof Throwable)
        appendException(new PrintWriter(writer), (Throwable) details);
    else if (details instanceof IStatus)
        appendStatus(new PrintWriter(writer), (IStatus) details, 0);
    text.setText(writer.toString());

    return text;
}

```

ExceptionDetailsDialog类在更一般的AbstractDetailsDialog类的最高级创建。该抽象对话框具有一个细节部分。该细节部分可以由用户选择是显示还是隐藏。而子类负责提供细节部分的内容。

```

public abstract class AbstractDetailsDialog extends Dialog
{
    private final String title;
    private final String message;
    private final Image image;

    public AbstractDetailsDialog(Shell parentShell, String title,
        Image image, String message)
    {
        this(new SameShellProvider(parentShell), title, image, message);
    }

    public AbstractDetailsDialog(IShellProvider parentShell,
        String title, Image image, String message)
    {
        super(parentShell);
        this.title = title;
        this.image = image;
        this.message = message;
        setShellStyle(SWT.DIALOG_TRIM | SWT.RESIZE
            | SWT.APPLICATION_MODAL);
    }

    configureShell()方法负责设置标题:
    protected void configureShell(Shell shell) {
        super.configureShell(shell);
        if (title != null)

```



```

        shell.setText(title);
    }

```

createDialogArea()方法创建并返回该对话框的上面部分的内容（在按钮栏以上）。它包括一个图像和一条消息（如果指定的话）。

```

protected Control createDialogArea(Composite parent) {
    Composite composite = (Composite) super.createDialogArea(parent);
    composite.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    if (image != null) {
        ((GridLayout) composite.getLayout()).numColumns = 2;
        Label label = new Label(composite, 0);
        image.setBackground(label.getBackground());
        label.setImage(image);
        label.setLayoutData(new GridData(
            GridData.HORIZONTAL_ALIGN_CENTER
            | GridData.VERTICAL_ALIGN_BEGINNING));
    }
    Label label = new Label(composite, SWT.WRAP);
    if (message != null)
        label.setText(message);
    GridData data = new GridData(GridData.FILL_HORIZONTAL
        | GridData.VERTICAL_ALIGN_CENTER);
    data.widthHint = convertHorizontalDLUsToPixels(
        IDialogConstants.MINIMUM_MESSAGE_AREA_WIDTH);
    label.setLayoutData(data);
    label.setFont(parent.getFont());

    return composite;
}

```

覆盖createButtonsForButtonBar()方法以创建OK和Details按钮。

```

private Button detailsButton;

protected void createButtonsForButtonBar(Composite parent) {
    createButton(parent, IDialogConstants.OK_ID,
        IDialogConstants.OK_LABEL, false);
    detailsButton = createButton(parent, IDialogConstants.DETAILS_ID,
        IDialogConstants.SHOW_DETAILS_LABEL, false);
}

```

当按下OK或Details按钮时，调用buttonPressed()方法。覆盖该方法以在按下Details按钮时交替显示或隐藏细节区域。

```

private Control detailsArea;
private Point cachedWindowSize;

protected void buttonPressed(int id) {
    if (id == IDialogConstants.DETAILS_ID)
        toggleDetailsArea();
    else
        super.buttonPressed(id);
}

protected void toggleDetailsArea() {
    Point oldWindowSize = getShell().getSize();

```

```

Point newWindowSize = cachedWindowSize;
cachedWindowSize = oldWindowSize;

// Show the details area.
if (detailsArea == null) {
    detailsArea = createDetailsArea((Composite) getContents());
    detailsButton.setText(IDialogConstants.HIDE_DETAILS_LABEL);
}

// Hide the details area.
else {
    detailsArea.dispose();
    detailsArea = null;
    detailsButton.setText(IDialogConstants.SHOW_DETAILS_LABEL);
}

/*
 * Must be sure to call
 *   getContents().computeSize(SWT.DEFAULT, SWT.DEFAULT)
 * before calling
 *   getShell().setSize(newWindowSize)
 * since controls have been added or removed.
 */
// Compute the new window size.
Point oldSize = getContents().getSize();
Point newSize = getContents().computeSize(
    SWT.DEFAULT, SWT.DEFAULT);
if (newWindowSize == null)
    newWindowSize = new Point(oldWindowSize.x, oldWindowSize.y
        + (newSize.y - oldSize.y));

// Crop new window size to screen.
Point windowLoc = getShell().getLocation();
Rectangle screenArea =
    getContents().getDisplay().getClientArea();
if (newWindowSize.y > screenArea.height
    - (windowLoc.y - screenArea.y))
    newWindowSize.y = screenArea.height
        - (windowLoc.y - screenArea.y);

getShell().setSize(newWindowSize);
((Composite) getContents()).layout();
}

```

最后，子类必须实现createDetailsArea()为对话框的区域提供内容。该区域当按下Details按钮被设为可见。

```
protected abstract Control createDetailsArea(Composite parent);
```

11.1.10 打开对话框——查找父shell

在创建新对话框时，你需要知道父shell或可以提供父shell的对象（具有getShell()方法或实现IShellProvider接口的对象）。你可以为父shell指定null，但这将阻止对话框与它的父部件的合适的关联。如果该对话框与其他许多的对话框同样的是模态，那么指定正确的父shell或shell提供者将阻止用户在关闭对话框之前激活父窗口。所以问题变成了：你如何获得父shell？

IHandler (参见6.3.1节) ——如果你有一个处理器, 那么你可以使用一个HandlerUtil方法和事件参数获取活动的shell。

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    Shell parentShell = HandlerUtil.getActiveShell(event);
    MyDialog dialog = new MyDialog(parentShell, ...);
    ... etc ...
}
```

IWorkbenchWindowActionDelegate (参见6.6.6节) ——如果你有一个操作代表, 那么Eclipse提供工作台窗口, 可以从中获取shell。在操作代表被初始化之后, Eclipse将立即调用init()方法, 并使用工作台窗口作为参数。缓存该窗口并在创建你的对话框时, 将窗口的shell作为参数传递。

```
private IWorkbenchWindow window;

public void init(IWorkbenchWindow window) {
    this.window = window;
}

public void run(IAction action) {
    Shell parentShell = window.getShell();
    MyDialog dialog = new MyDialog(parentShell, ...);
    ... etc ...
}
```

IObjectActionDelegate (参见6.7.3节) ——如果你在上下文菜单中有一个操作, Eclipse提供目标部分, 可以从中获取shell提供者。在调用run()方法之前, Eclipse使用目标部分调用setActivePart()。缓存该部分并在创建你的对话框时将包含该部分的站点作为参数传递。

```
private IWorkbenchPart targetPart;

public void setActivePart(IAction action, IWorkbenchPart targetPart)
{
    this.targetPart = targetPart;
}

public void run(IAction action) {
    IWorkbenchPartSite site = targetPart.getSite();
    MyDialog dialog = new MyDialog(site, ...);
    ... etc ...
}
```

IViewPart或IEditorPart (参见7.2节或8.2节) ——如果你有一个视图或编辑器, 与上面的代码类似, 你可以获取一个shell提供者:

```
IShellProvider shellProvider = viewOrEditor.getSite();
```

PlatformUI——平台UI提供工作台窗口, 可以从中获取shell。

```
Shell parentShell =
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell();
```

Display (参见4.2.5节) ——如果其他所有的方法都失败了, 你可以从Display获取活动窗口的shell。

```
Shell parentShell = Display.getDefault().getActiveShell();
```

11.2 向导

org.eclipse.jface.wizard.WizardDialog是Dialog的一个特殊子类 (图11-6)。当模态操作作为它的信

息集请求一个特殊序列时，或当一个屏幕有太多的字段时，使用它。向导在它的顶部有一个标题区域，如果需要，还可以有一个进度栏，以及在底部的Help、Next、Back、Finish和Cancel按钮（或一些子集）（图11-7）。标题区域包含了向导的标题、描述、一个可选的图像和一个错误、警告或需要的信息类消息。

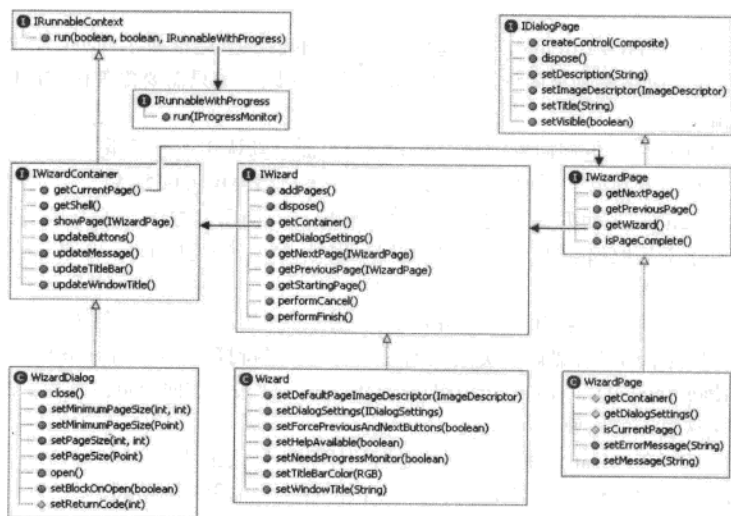


图11-6 向导类层次结构

11.2.1 IWizard

相对于继承 WizardDialog，你更应继承 org.eclipse.jface.wizard.Wizard。它实现 org.eclipse.jface.wizard.IWizard 接口以用于 WizardDialog。WizardDialog 使用 IWizard 接口以获取要显示的页面，并将用户操作通知向导。明确的向导类提供了大部分 IWizard 行为，允许你聚焦于 IWizard 接口的一个子集。向导的任务是创建和初始化它包含的页面，处理所有的特殊自定义流和页面间的信息，并在按下 Finish 按钮时执行操作。

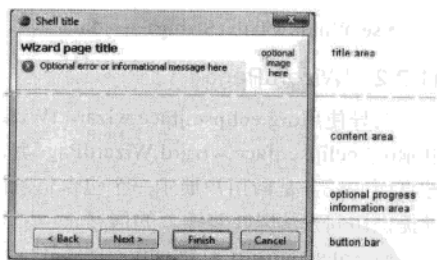


图11-7 默认向导对话框结构

- **addPages()**——子类应覆盖该方法以通过调用addPage()添加合适的页面。
- **canFinish()**——返回该向导是否可以在不需要进一步的用户交互下完成。一般地，该方法由向导容器使用以使Finish按钮可用或不可用。
- **createPageControls(Composite)**——在给定的父控件中创建该向导的控件。
- **dispose()**——清理所有本地资源，如图像、剪切板等由该类创建的资源。该方法遵循谁创建，谁销毁的Eclipse主题。
- **getContainer()**——返回显示该向导的向导容器。

- `getDefaultPageImage()`——返回该向导的默认页面图像。
- `getDialogSettings()`——返回该向导页面的对话框设置。
- `getNextPage(IWizardPage)`——返回将在指定向导页之后显示的页面。如果没有，返回`null`。默认实现以被添加至向导的顺序显示页面，所以子类只需要覆盖该方法以实现一个自定义的页面流。
- `getPreviousPage(IWizardPage)`——返回将于指定的向导页之前显示的页面。如果没有，返回`null`。默认实现以被添加至向导的顺序显示页面。所以子类只需要覆盖该方法以实现一个自定义的页面流。
- `getStartingPage()`——返回向导中显示的第一个页面。默认实现返回添加至向导的第一个页面，所以在开始页面不是第一个被添加的页面时，子类只需要覆盖该方法。
- `performCancel()`——如果向导被取消，将由向导容器调用该方法。子类只需要覆盖该方法以提供任意自定义的取消处理。如果向导容器可以被关闭，则返回`true`。如果它应继续保持打开，返回`false`。
- `performFinish()`——当按下Finish按钮时由向导容器调用。子类应覆盖该方法以执行向导操作并返回`true`以表示向导容器应被关闭。如果它应继续保持打开，则返回`false`。
- `setDefaultPageImageDescriptor(ImageDescriptor)`——如果当前向导页没有指定图像，设置在向导的标题区域显示的图像。
- `setHelpAvailable(boolean)`——设置帮助是否可用和Help按钮是否可见。
- `setNeedsProgressMonitor(boolean)`——设置该向导是否需要一个进度监视器。如果为`true`，那么在页面区域的下面，按钮的上面将保留空间以用于一个将显示的进度栏和进度消息（参见9.4节）。
- `setTitleBarColor(RGB)`——设置标题区域的颜色。
- `setWindowTitle(String)`——设置窗口标题。

11.2.2 IWizardPage

向导使用`org.eclipse.jface.wizard.IWizardPage`接口与它们包含的页面进行交流。一般地，你将会继承`org.eclipse.jface.wizard.WizardPage`类，访问和覆盖下列方法，而不是实现`IWizardPage`接口。向导页面的任务是向用户展示一个包含信息的页面，验证用户在该页面输入的信息的合法性，并为向导提供访问方法以收集输入的信息。

- `createControl(Composite)`——创建组成该向导页面的控件。
- `dispose()`——清理所有本地资源，如图像、剪切板等由该类创建的资源。该方法遵循谁创建，谁销毁的Eclipse主题。
- `getContainer()`——为该向导页返回向导容器。
- `getDialogSettings()`——为该向导页返回对话框设置。
- `getWizard()`——返回持有该向导页的向导。
- `setDescription(String)`——设置出现于向导标题区域的描述文本。
- `setErrorMessage(String)`——为该页设置或消除错误消息。
- `setImageDescriptor(ImageDescriptor)`——设置出现于向导的标题区域的图像。
- `setMessage(String)`——为该页设置或清除消息。

- `setPageComplete(boolean)`——设置当前页是否已完成。该方法是决定Next和Finish按钮是否可用的前提。
- `setTitle(String)`——设置出现于向导标题区域的标题，而不是在shell内的标题。
- `setVisible(boolean)`——设置该对话框页的可见性。子类可以扩展该方法（要确认调用超类方法）以检测一个页面什么时候成为活动页面。

验证用户在向导页中输入信息的合法性的一种方法是为该向导页中的每一个字段提供一个调用`updatePageComplete()`方法的监听器（从11.2.8节中获取）。

```
sourceFileField = new Text(container, SWT.BORDER);
sourceFileField.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        updatePageComplete();
    }
});
```

该`updatePageComplete()`方法将负责检查每个字段的内容，在恰当的时候显示一条错误消息，并调用`setPageComplete()`方法（参见11.2.8节以了解示例）。`pageComplete`属性被向导容器用于决定Next和Finish按钮是否应被设为可用。

从Eclipse 3.2开始的新特性 `org.eclipse.jface.fieldassist`包添加图像、悬停文本、内容建议，并自动完成表单、对话框和向导中的字段。该功能可以帮助用户理解并输入信息至基于表单的用户界面。

11.2.3 IWizardContainer

向导使用`org.eclipse.jface.wizard.IWizardContainer`接口与将要显示它的上下文进行交流。

- `getCurrentPage()`——返回当前显示的页面。
- `getShell()`——返回该向导容器的shell。
- `run(boolean, boolean, IRunnableWithProgress)`——运行向导对话框的上下文中给定的可运行项。第一个参数“分叉”（fork）表示可执行项是否应在一个单独线程中执行。第二个参数“可撤销”（cancelable）表示是否允许用户在执行过程中撤销操作（参见4.2.5节以了解更多关于UI线程的内容）。
- `showPage(IWizardPage)`——显示指定的向导页。该方法不应用于一般的下一步/上一步（next/back）页面流，而是为了诸如在列表中双击之类的自定义页面流而存在。
- `updateButtons()`——调整Back、Next和Finish按钮的可用状态以反映该容器中的活动页面的状态。
- `updateMessage()`——更新消息栏中显示的消息，以反映该容器中当前活动页的状态。
- `updateTitleBar()`——更新标题栏（标题、描述和图像），以反映该容器中活动页的状态。
- `updateWindowTitle()`——更新窗口标题栏，以反映向导的状态。

除了`IWizardContainer`之外，`WizardDialog`还实现了`IWizardContainer2`和`IPageChangeProvider`。在你的向导中，你可以使用`getContainer()`获取容器，然后测试它以查看是否实现了这些接口以访问附加功能。

- `addPageChangedListener(IPageChangeListener)`——为该页面更改提供者中的页面更改添加一个监听器。
- `getSelectedPage()`——返回对话框中的当前选中页。

- `removePageChangeListener(IPageChangeListener)`——从该页面更改提供者移除给定页面更改监听器。
- `updateSize()`——更新窗口大小，以反映当前向导的状态。

11.2.4 嵌套的向导

向导可以包含一个或多个嵌套向导，如Import和Export向导。`org.eclipse.jface.wizard.WizardSelectionPage`类向用户提供管理一个或多个嵌套向导的行为。当可以确定嵌套向导时，一个`WizardSelectionPage`子类将调用方法。当用户点击Next按钮时，`WizardSelectionPage`类将通过`org.eclipse.jface.wizard.IWizardNode`接口使用该信息以创建和管理嵌套向导。

11.2.5 启动向导

你可以使用一个预定义向导扩展点将向导关联至Eclipse框架，或者作为用户操作的结果，你可以手动启动向导。

1. 向导扩展点

如果你想要Eclipse自动提供操作代表并在预定位置显示你的向导，你可以扩展以下向导扩展点之一。

`org.eclipse.ui.exportWizards`——在导出向导中添加一个嵌套向导，可以通过选择File > Export...命令显示它。与该扩展点关联的向导类必须实现`org.eclipse.ui.IExportWizard`接口。

`org.eclipse.ui.importWizards`——在导入向导中添加一个嵌套向导。可以通过选择File > Import...显示它。与该扩展点关联的向导类必须实现`org.eclipse.ui.IImportWizard`接口。

`org.eclipse.ui.newWizards`——在新建向导中添加一个嵌套向导。可以通过选择File > New > Other...显示它。与该扩展点关联的向导类必须实现`org.eclipse.ui.INewWizard`接口。

这三个扩展点共享几个通用属性：

- `class`——将由父向导启动的向导类。该向导必须为之前列出的扩展点实现合适的接口。该类使用它的无参数构造函数进行初始化，但可以通过使用`IExecutableExtension`接口赋予参数（参见21.5节）。
- `icon`——与该向导关联的图标，类似于操作的图像（参见6.6.4节）。
- `id`——向导的唯一标识符。
- `name`——向导的可读名称。

你可以使用selection子元素为`org.eclipse.ui.exportWizards`和`org.eclipse.ui.importWizards`扩展点指定过滤程序。通过使用过滤程序，你的导出或导入向导仅当恰当命名或被指定类型的元素被选中时才显示。

`org.eclipse.ui.newWizards`扩展点需要一个额外的category属性以指定向导是如何被组织成层次结构的。该category使用具有以下属性的同一个扩展点进行声明：

- `id`——该类别的唯一标识符。
- `name`——该类别的可读名称。
- `parentCategory`——将显示该类别的上级类别的唯一标识符（如果存在）。

`org.eclipse.ui.newWizards`扩展点也允许你指定一个primaryWizard。主向导将在新建的向导对话框中加粗显示并存在，以使产品管理者可以强调它们产品的向导集。该元素不是为插件开发者准备的。

如果使用上述扩展点之一声明的向导实现了IExecutableExtension接口,那么Eclipse将与使用该接口的向导交流声明中编码的附加初始化信息(参见21.5节)。

你可以使用插件清单编辑器快速创建具有与向导扩展点关联的存根方法的向导类。在插件清单编辑器中,浏览至Extensions页面并点击Add...按钮以添加,比如,一个org.eclipse.ui.newWizards扩展项(参见6.6.1节以了解关于添加扩展项的示例)。然后,右键点击插件清单编辑器中的org.eclipse.ui.newWizards扩展项,并选择New > wizard。选择被添加的向导扩展项以编辑它的属性(图11-8)。

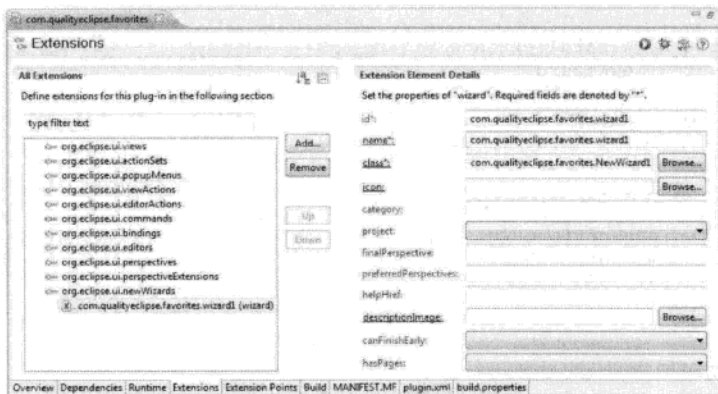


图11-8 显示newWizard扩展项的扩展项元素细节

点击class字段右侧的Class:标签将打开一个New Java Class向导。在该向导中可以创建新的类(图11-9)。

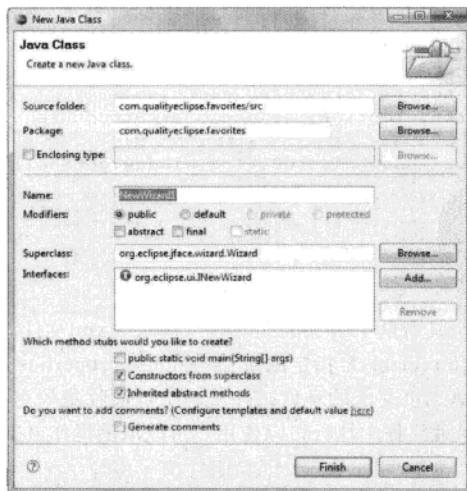


图11-9 显示创建类的新建Java类向导

填充所需字段将生成一个新的Java类并将该类关联至插件清单。下列内容是生成的插件清单条目和存根Java向导类。

```
<extension point="org.eclipse.ui.newWizards">
    <wizard
        name="com.qualityeclipse.favorites.wizard1"
        class="com.qualityeclipse.favorites.NewWizard1"
        id="com.qualityeclipse.favorites.wizard1">
    </wizard>
</extension>

public class NewWizard1 extends Wizard
    implements INewWizard
{
    public void init(
        IWorkbench workbench, IStructuredSelection selection
    ) {
        // Initialization code here.
    }
    public boolean performFinish() {
        // Perform operation here.
        return false;
    }
}
```

2. 手动启动向导

作为替代，你也可以从命令处理器启动向导。在这种情况下，实现IHandler或扩展具有execute方法的AbstractHandler（参见6.3.1节）。该execute方法启动获取字符串（Extract Strings）向导（参见下一节）。

```
public Object execute(ExecutionEvent event) throws ExecutionException {
    IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindow(event);
    ISelection selection = HandlerUtil.getCurrentSelection(event);

    ExtractStringsWizard wizard = new ExtractStringsWizard();
    wizard.init(window.getWorkbench(),
        selection instanceof IStructuredSelection
            ? (IStructuredSelection) selection : StructuredSelection.EMPTY);

    WizardDialog dialog = new WizardDialog(window.getShell(), wizard);
    dialog.open();
    return null;
}
```

11.2.6 向导示例

作为示例，你将创建一个具有两个页面的向导，从plugin.xml文件获取字符串，并将这些字符串放置到一个由《RFRS Requirements》指定的单独的plugin.properties文件。该向导负责初始化这两个页面，降低从第一页至第二页交流的难度，并从两个页面收集信息，然后在用户按下Finish按钮时执行操作。该操作将在单独线程中执行，以使用户可以取消操作（参见9.4节和4.2.5节以了解更多关于UI线程的内容）。

在下面的代码示例中，操作代表直接调用init()方法，而当向导对话框被创建和打开时，对话框框架简介调用addPages()方法。这种方法与INewWizard接口并行，这样该向导可以简单的实现该接

口并因此可以通过File > New > Other...启动。

```
public class ExtractStringsWizard extends Wizard
    implements INewWizard
{
    private IStructuredSelection initialSelection;
    private SelectFilesWizardPage selectFilesPage;
    private SelectStringsWizardPage selectStringsPage;
    public void init(
        IWorkbench workbench, IStructuredSelection selection
    ) {
        initialSelection = selection;
    }
    public void addPages() {
        setWindowTitle("Extract");
        selectFilesPage = new SelectFilesWizardPage();
        addPage(selectFilesPage);
        selectStringsPage = new SelectStringsWizardPage();
        addPage(selectStringsPage);
        selectFilesPage.init(initialSelection);
    }
}
```

当用户完成输入信息并点击Finish按钮时，将调用performFinish()方法执行操作。在该示例中，finish方法使用由向导容器提供的功能在单独线程中执行操作，这样UI在执行期间保持对用户的响应，因此用户可以撤销操作。

```
public boolean performFinish() {
    final ExtractedString[] extracted =
        selectStringsPage.getSelection();

    try {
        getContainer().run(true, true, new IRunnableWithProgress() {
            public void run(IProgressMonitor monitor)
                throws InvocationTargetException, InterruptedException
            {
                performOperation(extracted, monitor);
            }
        });
    }
    catch (InvocationTargetException e) {
        FavoritesLog.logError(e);
        return false;
    }
    catch (InterruptedException e) {
        // User canceled, so stop but don't close wizard.
        return false;
    }
    return true;
}

private void performOperation(
    ExtractedString[] extracted, IProgressMonitor monitor
) {
    monitor.beginTask("Extracting Strings", extracted.length);
    for (int i = 0; i < extracted.length; i++) {
        // Replace sleep with actual work
        Thread.sleep(1000);
    }
}
```

```

        if (monitor.isCanceled())
            throw new InterruptedException("Canceled by user");
        monitor.worked(1);
    }
    monitor.done();
}

```

11.2.7 对话框设置

对话框设置可用于存储向导或对话框的当前值，以待下次打开该向导或对话框时使用。在这种情况下，在向导的构造函数中初始化并缓存对话框设置对象，由不同的向导页面使用。getSection()调用被用于隔离不同向导的设置。每一个页面随后可以使用不同的IDialogSetting get()和put()方法在会话间载入和保存值。

```

public ExtractStringsWizard() {
    IDialogSettings favoritesSettings =
        FavoritesActivator.getDefault().getDialogSettings();
    IDialogSettings wizardSettings =
        favoritesSettings.getSection("ExtractStringsWizard");
    if (wizardSettings == null)
        wizardSettings =
            favoritesSettings.addNewSection("ExtractStringsWizard");
    setDialogSettings(favoritesSettings);
}

```

11.2.8 基于选择的页面内容

获取字符串 (Extract Strings) 向导的第一个页面显示源文件 (Source File) 和目标文件 (Destination File) 文本字段，都在右侧具有一个Browse...按钮 (图11-10)。createControl()方法创建并对齐向导页的控件。

```

public class SelectFilesWizardPage extends WizardPage
{
    private Text sourceFileField;
    private Text destinationFileField;
    private IPath initialSourcePath;
    public SelectFilesWizardPage() {
        super("selectFiles");
        setTitle("Select files");
        setDescription(
            "Select the source and destination files");
    }

    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NULL);
        final GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 3;
        container.setLayout(gridLayout);
        setControl(container);

        final Label label = new Label(container, SWT.NONE);
        final GridData gridData = new GridData();
        gridData.horizontalSpan = 3;
        label.setLayoutData(gridData);
    }
}

```

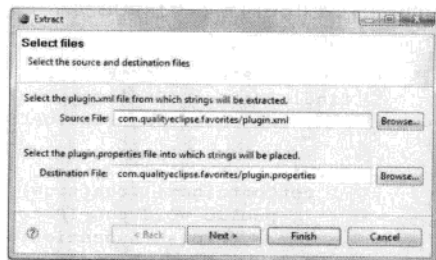


图11-10 获取字符串向导

```
label.setText(
    "Select the plugin.xml file " +
    "from which strings will be extracted.");

final Label label_1 = new Label(container, SWT.NONE);
final GridData gridData_1 =
    new GridData(GridData.HORIZONTAL_ALIGN_END);
label_1.setLayoutData(gridData_1);
label_1.setText("Source File:");

sourceFileField = new Text(container, SWT.BORDER);
sourceFileField.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        updatePageComplete();
    }
});
sourceFileField.setLayoutData(
    new GridData(GridData.FILL_HORIZONTAL));

final Button button = new Button(container, SWT.NONE);
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        browseForSourceFile();
    }
});
button.setText("Browse...");

final Label label_2 = new Label(container, SWT.NONE);
final GridData gridData_2 = new GridData();
gridData_2.horizontalSpan = 3;
label_2.setLayoutData(gridData_2);

final Label label_3 = new Label(container, SWT.NONE);
final GridData gridData_3 = new GridData();
gridData_3.horizontalSpan = 3;
label_3.setLayoutData(gridData_3);
label_3.setText(
    "Select the plugin.properties file " +
    "into which strings will be placed.");
final Label label_4 = new Label(container, SWT.NONE);
final GridData gridData_4 = new GridData();
gridData_4.horizontalIndent = 20;
label_4.setLayoutData(gridData_4);
label_4.setText("Destination File:");

destinationFileField =
    new Text(container, SWT.BORDER);
destinationFileField.addModifyListener(
    new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            updatePageComplete();
        }
    });
destinationFileField.setLayoutData(
    new GridData(GridData.HORIZONTAL_ALIGN_FILL));

final Button button_1 =
    new Button(container, SWT.NONE);
```

```

        button_1
            .addSelectionListener(new SelectionAdapter() {
                public void widgetSelected(SelectionEvent e) {
                    browseForDestinationFile();
                }
            });
        button_1.setText("Browse...");

        initContents();
    }
}

```

同往常一样，我们致力于为用户节省时间。如果用户在工作台中已经选择了一些项，你将打算根据这些信息生成向导页。对于该向导页而言，init()方法分析当前选择并缓存结果。而initContents()方法根据该缓存结果初始化字段内容。

```

public void init(IStructuredSelection selection) {
    if (selection == null)
        return;
    // Find the first plugin.xml file.
    Iterator<?> iter = selection.iterator();
    while (iter.hasNext()) {
        Object item = iter.next();
        if (item instanceof IJavaElement) {
            IJavaElement javaElem = (IJavaElement) item;
            try {
                item = javaElem.getUnderlyingResource();
            }
            catch (JavaModelException e) {
                // Log and report the exception.
                e.printStackTrace();
                continue;
            }
        }
        if (item instanceof IFile) {
            IFile file = (IFile) item;
            if (file.getName().equals("plugin.xml")) {
                initialSourcePath = file.getLocation();
                break;
            }
            item = file.getProject();
        }
        if (item instanceof IProject) {
            IFile file = ((IProject) item).getFile("plugin.xml");
            if (file.exists()) {
                initialSourcePath = file.getLocation();
                break;
            }
        }
    }
}

private void initContents() {
    if (initialSourcePath == null) {
        setPageComplete(false);
        return;
    }
    IPath rootLoc = ResourcesPlugin.getWorkspace()

```

```
.getRoot().getLocation();
IPath path = initialSourcePath;
if (rootLoc.isPrefixOf(path))
    path = path
        .setDevice(null)
        .removeFirstSegments(rootLoc.segmentCount());
sourceFileField.setText(path.toString());
destinationFileField.setText(
    path
        .removeLastSegments(1)
        .append("plugin.properties")
        .toString());
updatePageComplete();
}
```

向导在标题下面提供了一个可以显示反馈的消息区域，该区域用于向用户显示在前进至下一向导页或执行操作需要用户输入的附加信息。在这种情况下，updatePageComplete()方法在确定了初始内容之后又一次被不同的文本字段监听器在内容被更改时调用。该方法随后监听当前文本字段的内容，显示错误或警告消息，并恰当地将Next和Finish按钮设为可用或不可用。

```
private void updatePageComplete() {
    setPageComplete(false);
    IPath sourceLoc = getSourceLocation();
    if (sourceLoc == null || !sourceLoc.toFile().exists()) {
        setMessage(null);
        setErrorMessage("Please select an existing plugin.xml file");
        return;
    }
    IPath destinationLoc = getDestinationLocation();
    if (destinationLoc == null) {
        setMessage(null);
        setErrorMessage(
            "Please specify a plugin.properties file"
            + " to contain the extracted strings");
        return;
    }
    setPageComplete(true);

    IPath sourceDirPath = sourceLoc.removeLastSegments(1);
    IPath destinationDirPath = destinationLoc.removeLastSegments(1);
    if (!sourceDirPath.equals(destinationDirPath)) {
        setErrorMessage(null);
        setMessage(
            "The plugin.properties file is typically"
            + " located in the same directory"
            + " as the plugin.xml file",
            WARNING);
        return;
    }

    if (!destinationLoc.lastSegment().equals("plugin.properties")) {
        setErrorMessage(null);
        setMessage(
            "The destination file is typically"
            + " named plugin.properties",
            WARNING);
    }
}
```

```

        return;
    }

    setMessage(null);
    setErrorMessage(null);
}

```

当用户点击Browse按钮时，选择监听器调用browseForSourceFile()方法以提示用户选择源文件。你还需要一个名为browseForDestinationFile()的类似方法。该方法在点击其他Browse按钮时调用。同时，你还需要源文件和目标文件位置的访问方法。

```

protected void browseForSourceFile() {
    IPath path = browse(getSourceLocation(), false);
    if (path == null)
        return;
    IPath rootLoc = ResourcesPlugin.getWorkspace()
        .getRoot().getLocation();
    if (rootLoc.isPrefixOf(path))
        path = path.setDevice(null)
            .removeFirstSegments(rootLoc.segmentCount());
    sourceFileField.setText(path.toString());
}

private IPath browse(IPath path, boolean mustExist) {
    FileDialog dialog = new FileDialog(getShell(),
        mustExist ? SWT.OPEN : SWT.SAVE);
    if (path != null) {
        if (path.segmentCount() > 1)
            dialog.setFilterPath(path.removeLastSegments(1)
                .toOSString());
        if (path.segmentCount() > 0)
            dialog.setFileName(path.lastSegment());
    }
    String result = dialog.open();
    if (result == null)
        return null;
    return new Path(result);
}

public IPath getSourceLocation() {
    String text = sourceFileField.getText().trim();
    if (text.length() == 0)
        return null;
    IPath path = new Path(text);
    if (!path.isAbsolute())
        path = ResourcesPlugin.getWorkspace().getRoot().getLocation()
            .append(path);
    return path;
}

```

11.2.9 基于前一页面的页面内容

向导的第二页包含了一个名/值对的单选框列表。这些名/值对可以从源文件中获取（图 11-11）。

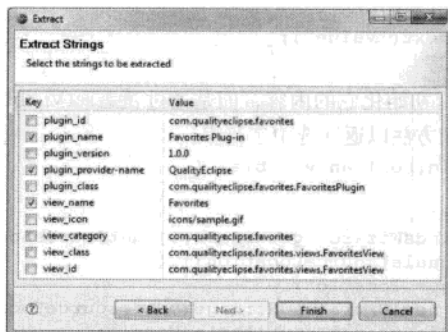


图11-11 获取字符串向导的第二页

```
public class SelectStringsWizardPage extends WizardPage
{
    private CheckboxTableViewer checkboxTableViewer;
    private IPath sourceLocation;
    private ExtractedStringsModel stringModel;

    public SelectStringsWizardPage() {
        super("selectStrings");
        setTitle("Extract Strings");
        setDescription("Select the strings to be extracted");
    }

    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NULL);
        container.setLayout(new FormLayout());
        setControl(container);

        checkboxTableViewer =
            CheckboxTableViewer.newCheckList(container, SWT.BORDER);
        checkboxTableViewer.setContentProvider(
            new ExtractedStringsContentProvider());
        checkboxTableViewer.setLabelProvider(
            new ExtractedStringsLabelProvider());

        final Table table = checkboxTableViewer.getTable();
        final FormData formData = new FormData();
        formData.bottom = new FormAttachment(100, 0);
        formData.right = new FormAttachment(100, 0);
        formData.top = new FormAttachment(0, 0);
        formData.left = new FormAttachment(0, 0);
        table.setLayoutData(formData);
        table.setHeaderVisible(true);

        final TableColumn tableColumn =
            new TableColumn(table, SWT.NONE);
        tableColumn.setWidth(200);
        tableColumn.setText("Key");

        final TableColumn tableColumn_1 =
            new TableColumn(table, SWT.NONE);
```



```

        tableColumn_1.setWidth(250);
        tableColumn_1.setText("Value");
    }

```

该页不是在第一次创建时初始化它的内容，而是通过覆盖setVisible()方法在它变成可见时更新它的内容。你还需要一个访问方法以返回选中字符串。

```

public void setVisible(boolean visible) {
    if (visible) {
        IPath location =
            ((ExtractStringsWizard) getWizard()).getSourceLocation();
        if (!location.equals(sourceLocation)) {
            sourceLocation = location;
            stringModel = new ExtractedStringsModel(sourceLocation);
            checkboxTableViewer.setInput(stringModel);
        }
    }
    super.setVisible(visible);
}

public ExtractedString[] getSelection() {
    Object[] checked = checkboxTableViewer.getCheckedElements();
    int count = checked.length;
    ExtractedString[] extracted = new ExtractedString[count];
    System.arraycopy(checked, 0, extracted, 0, count);
    return extracted;
}

```

还有两个模型类，ExtractedString和ExtractedStringsModel，和两个查看器帮助类，与本书稍早已经讨论过的类似，ExtractedStringsContentProvider和ExtractedStringsLabelProvider。这些类可以在QualityEclipse网站（www.qualityeclipse.com）示例代码中找到。为了了解更多关于这些类型的类，参见以下内容：

- 7.2.3节，视图模型
- 7.2.4节，内容提供者
- 7.2.5节，标签提供者

11.3 RFRS相关事项

《RFRS Requirements》的“用户界面”一节包含了5个与向导相关的内容（4个要求和1个最佳做法）。它们大部分都来源于Eclipse UI准则。

11.3.1 向导外观 (RFRS 3.5.2)

用户界面准则#5.2是一个要求。它说明：

每一个向导都必须包含一个具有标题图形和一个文本区域的头部以用于用户反馈。它还必须在页脚包含Back、Next、Finish和Cancel按钮。一个只有一页的向导不需要具有Back和Next按钮。

展示你的向导遵循标准向导外观。确认它们包含正确排序的合适的按钮，以及一个合适的标题图形。

11.3.2 在编辑器中打开新文件 (RFRS 3.5.6)

用户界面准则#5.9是一个要求。它说明：

如果创建了一个新文件，在编辑器中打开该文件。如果创建了一组文件，在编辑器中打开最重要的文件或中心文件。

如果你的向导创建了一个文件，展示它当向导完成时自动在一个编辑器中打开。对于获取字符串向导来说，你应展示plugin.properties文件在向导创建它之后打开。

11.3.3 新项目切换透视图 (RFRS 3.5.7)

用户界面准则#5.10是一个要求。它说明：

如果创建了一个新项目，提示用户并更改活动透视图以匹配项目类型。

如果你的插件提供了一个新项目向导和一个相关联的透视图，展示当你的向导被用于创建一个新项目时，系统自动切换至你的透视图。

11.3.4 显示新对象 (RFRS 3.5.8)

用户界面准则#5.11是一个要求。它说明：

如果创建一个单独的新对象，选择该对象并在合适的视图中显示它。

在资源的创建导致项目或文件夹资源的创建时，向导应建议合理的默认位置。

如果你的向导创建了文件，展示它自动的在合适视图中被选择。对于获取字符串向导来说，你应展示plugin.properties文件在向导创建它之后在导航视图中被选中。

11.3.5 单一页面向导按钮 (RFRS 5.3.5.13)

最佳做法#1说明：

一个单一页面向导必须包含Finish和Cancel按钮，还应包含不可用的Back和Next按钮。

如果你的插件包含任意单一页面向导，展示它包含合适状态的合适按钮。

11.4 总结

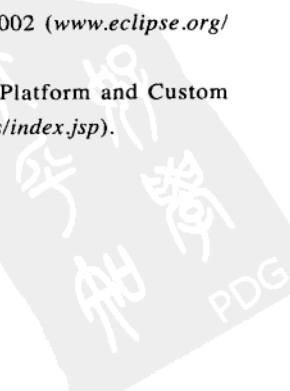
本章介绍了一些将在Eclipse创建开发过程中遇到的普通SWT和JFace对话框类。当一个内建的对话框或向导不能满足你的需要时，你可以使用本章描述的技术创建你自己的对话框或向导。

参考文献

本书资源 (2.9节).

Klinger, Doina, "Creating JFace Wizards," IBM UK, December 16, 2002 (www.eclipse.org/articles/Article-JFace%20Wizards/wizardArticle.html).

Fatima, Azra, "Wizards in Eclipse: An Introduction to Working with Platform and Custom Wizards," HP, August 2003 (devresource.hp.com/drc/technical_articles/wizards/index.jsp).



第12章 首选项页

大部分Eclipse插件提供了可由用户配置的首选项。这些首选项可以用于控制插件如何执行与显示信息。首选项框架提供了一种机制。这种机制用于向用户显示这些选项并在多个Eclipse会话间保存值。本章讨论如何创建一个Eclipse首选项页，以及用于记录和存储插件的首选项的技术。

12.1 创建首选项页

你需要添加一个首选项页。该页将允许用户选择对于Favorites产品可见的列。为了完成该任务，在插件清单中创建一个org.eclipse.ui.preferencePages扩展项。幸运的是，Eclipse提供了一个用于创建首选项页的向导。

打开Favorites plugin.xml文件并切换至Extensions页。点击Add...按钮以打开New Extensions向导，从扩展点列表中选择org.eclipse.ui.preferencePages，从模板列表中选择Preference Page，然后点击Next（图12-1）。在以下页，分别修改Page Class Name和Page Name为“FavoritesPreferencePage”和“Favorites”（图12-2），然后点击Finish。

点击Extensions页中的org.eclipse.ui.preferencePages扩展项中的新Favorites（page）扩展项，以显示它的属性。在扩展元素细节区域，你将看到Eclipse在线帮助描述的以下属性。更改id属性为“com.qualityeclipse.favorites.prefs.view”。

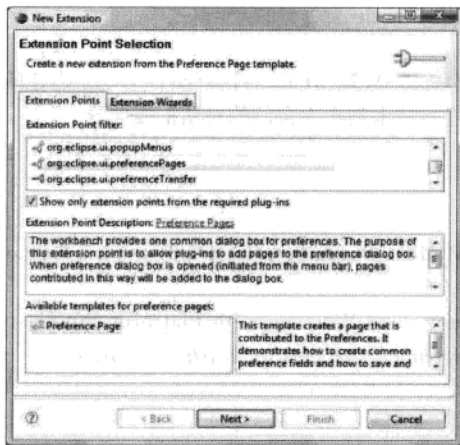


图12-1 新建扩展项向导

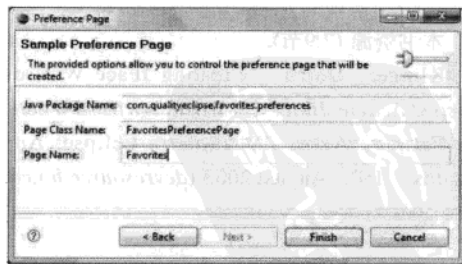


图12-2 示例首选项页向导

- id——将被用于标识该页的唯一名称。
- name——在首选项页的层次结构中显示的可读名称。该层次结构出现于工作台首选项对话框。
- class——实现org.eclipse.ui.IWorkbenchPreferencePage接口类的完全合格名称。该类使用它的

无参数构造函数初始化,但可以使用IExecutableExtension接口赋予参数(参见21.5节)。

- **category**——首选项对话框框提供了页面的一个层次结构组织。基于该原因,页面可以选择是否指定一个category属性。该属性表示一个由被“/”隔开的父页面ID组成的路径。如果忽略了该属性或无法找到路径中任意的父节点,页面将会被添加至根级别(参见12.2.6节)。

如果你启动Runtime Workbench,打开工作台Preferences对话框,选择Favorites,你将看到由New Extension向导(图12-3)创建的示例Favorites首选项页。

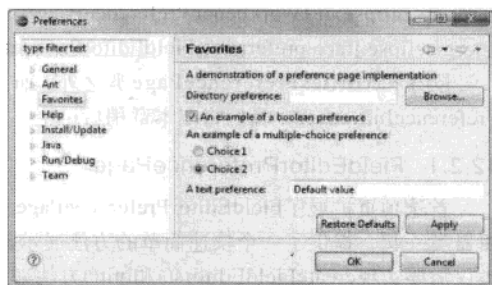


图12-3 示例收藏夹首选项页

12.2 首选项页API

在修改首选项页以达到我们的目的之前,请查看向导生成的内容(图12-4)。根据上一节列出的

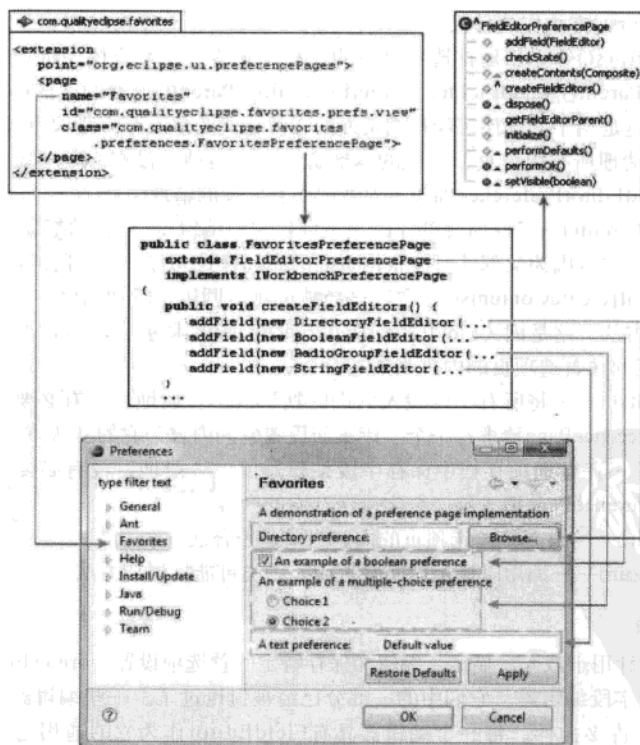


图12-4 首选项页声明、类和表示

内容, 插件清单包含了名称、标识符和定义页面的内容与行为的类的完全合格名称。首选项页必须实现org.eclipse.ui.IWorkbenchPreferencePage接口, 抽象类org.eclipse.jface.preference.PreferencePage和org.eclipse.jface.preference.FieldEditorPreferencePage, 为该目的提供大部分的基础结构。

除了FavoritesPreferencePage类之外, 向导还创建了两个其他的类, PreferenceConstants和PreferenceInitializer。我们将在本章稍后讨论。

12.2.1 FieldEditorPreferencePage





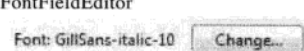

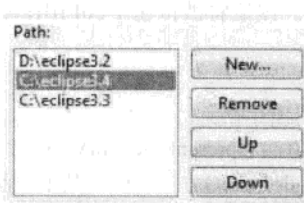
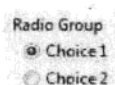
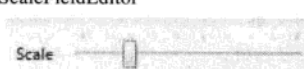
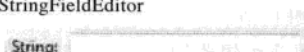
首选项页扩展了FieldEditorPreferencePage。该类与org.eclipse.jface.preference.*包中的不同的编辑器类一起, 提供了一个快速简单的方法展示和捕捉简单的首选项。FieldEditorPreferencePage的子类仅需要实现createFieldEditors()和init()方法来显示一个简单的首选项页。然而, 以下是你使用更复杂的首选项页时需要关注的另外几个方法。

- addField(FieldEditor)——从createFieldEditors()方法调用以添加一个字段至页面。
- checkState()——由FieldEditorPreferencePage调用以验证页面内容。该方法的FieldEditorPreferencePage实现请求每一个字段验证它的内容并用验证结果调用setValid()。覆盖该方法以执行额外的页面验证。
- createContents(Composite)——创建包含字段编辑器复合组件。一般地, 子类覆盖createFieldEditors()方法作为替代。
- createFieldEditors()——创建首选项页中的字段编辑器。子类应该为每一个创建的字段调用getFieldEditorParent()和addField()。由getFieldEditorParent()返回的父部件不应被用于超过一个的编辑器。这是由于父部件可能根据页面的布局样式为每一个字段编辑器而改变。
- dispose()——清理所有由该页面分配的本地资源。一般地, 没有必要覆盖该方法。这是因为dispose()的FieldEditorPreferencePage实现处理所有字段的清理。
- getFieldEditorParent()——返回父部件以用于创建字段编辑器。返回的父部件不应被用于超过一个的编辑器。这是因为父部件可能根据页面的布局样式为每一个字段编辑器而改变。
- initialize()——由createContents()在字段已经被创建后调用, 以初始化字段内容。一般地, 没有必要覆盖该方法。这是因为FieldEditorPreferencePage请求每一个字段初始化它自身。
- isValid()——返回该首选项页的内容当前是否合法。
- performDefaults()——将所有字段载入它们的默认值。一般地, 没有必要覆盖该方法, 因为FieldEditorPreferencePage请求每一个字段重新设置它的内容为它的默认值。
- performOk()——在首选项存档中保存字段编辑器值。一般地, 没有必要覆盖该方法, 因为FieldEditorPreferencePage请求每一个字段保存它的内容。
- setValid(boolean)——设置该首选项页的内容当前是否合法。
- setVisible(boolean)——调用以显示或隐藏页面。子类可能扩展该方法。

12.2.2 字段编辑器

字段编辑器被设计用于载入、显示、编辑和保存特定的首选项设置。org.eclipse.jface.preference包提供了许多不同的字段编辑器。它们中的一部分已经被讨论过了。一些编辑器包含一个控件, 而其他编辑器则包含了许多控件。每一个编辑器具有FieldEditor作为它的通用超类, 为FieldEditorPreferencePage提供了一种通用方法以访问编辑器。表12-1显示了一个org.eclipse.jface.preference包中的字段编辑器的列表。该列表还包括Eclipse中其他作为公用API的可用项。

表12-1 PreferencePage字段编辑器

字段编辑器	描 述
<p>BooleanFieldEditor</p> 	一个表示布尔值的单选框
<p>ColorFieldEditor</p> 	一个标签和按钮，按钮显示颜色首选项并在点击时打开一个颜色选择器
<p>DirectoryFieldEditor</p> 	一个标签、文本字段和按钮，用于选择目录。按钮在点击时打开一个目录选择器
<p>FileFieldEditor</p> 	一个标签、文本字段和按钮，用于选择一个文件首选项。按钮在点击时打开一个文件选择器。编辑器可以选择强制使用绝对文件路径和对于指定文件扩展名的过滤器
<p>FontFieldEditor</p> 	一个标签、字体名称和按钮，用于选择字体。按钮在点击时打开一个字体选择器
<p>IntegerFieldEditor</p> 	一个标签和文本字体，用于选择整数。该编辑器可以选择是否强制使用一个范围内的值
<p>PathEditor</p> 	一个标签和按钮组，用于选择0个或以上路径。New...按钮打开一个目录选择器，而其他按钮操作列表中已有路径
<p>RadioGroupFieldEditor</p> 	一个标签和单选按钮序列，用于选择几个属性之一。可选择地，单选按钮可以被分组并在多列中显示
<p>ScaleFieldEditor</p> 	一个标签和滑动栏，用于选择一个整数值范围
<p>StringFieldEditor</p> 	一个标签和文本字段，用于输入一个字符串值

字段编辑器基于“创建，然后遗忘”的主题设计。换句话说，你创建一个字段编辑器。该字段编辑器具有它将展示的首选项的所有相关内容。然后该字段编辑器，与FieldEditorPreferencePage一起，处理余下事项。

字段编辑器擅长于展示和操作简单类型的首选项，如字符串、整数、颜色等。如果你的首选项

大部分是这些类型的,那么字段编辑器将为你减少编写代码以载入、显示、验证和存储这些简单首选项的难度。如果你打算展示的数据是更加结构化和复杂的,那么你可能需要不使用字段编辑器来创建你的首选项页,继承PreferencePage而不是FieldEditorPreferencePage。如果你需要与一个字段编辑器直接交互,或创建一种新类型的字段编辑器,这里有一些字段编辑器方法是你可能需要了解的:

- `adjustForNumColumns(int)`——调整字段编辑器的基本控件的水平区域。
- `dispose()`——清理所有由该编辑器分配的本地资源。
- `doFillIntoGrid(Composite, int)`——创建组成编辑器的控件。
- `doLoad()`——使用从首选项存档中获取的当前值初始化编辑器内容。
- `doLoadDefault()`——使用默认值初始化编辑器内容。
- `doStore()`——保存当前编辑器值至首选项存档。
- `fireStateChanged(String, boolean, boolean)`——通知字段编辑器的监听器(如果存在),发生了一个布尔值属性的变更。如果旧值和新值是一样的,则不完成任何操作。
- `fireValueChanged(String, Object, Object)`——通知字段编辑器的监听器(如果存在),发生了一个属性的更改。
- `getLabelControl()`——返回组成编辑器的标签,如果没有,则返回null。
- `getLabelControl(Composite)`——返回组成编辑器的标签。如果标签文本已经在构造函数或`setLabelText()`方法中指定,则创建标签。
- `getLabelText()`——返回在构造函数或`setLabelText()`方法中指定的标签文本。
- `getNumberOfControls()`——返回组成编辑器的控件的数量。该值将被传递至`doFillIntoGrid(Composite, int)`方法。
- `getPreferenceName()`——返回编辑器显示的首选项的名/值对。
- `getPreferenceStore()`——返回包含当前被编辑的首选项的首选项存档。
- `isValid()`——返回编辑器的内容是否是合法的。子类应覆盖该方法和`presentsDefaultValue()`方法。
- `load()`——从首选项存档中载入当前值至编辑器。子类应覆盖`doLoad()`方法,而不是本方法。
- `loadDefault()`——载入默认值至编辑器。子类应覆盖`doLoadDefault()`方法而不是本方法。
- `presentsDefaultValue()`——返回编辑器是否当前正显示默认值。
- `refreshValidState()`——决定编辑器的内容是否是合法的。子类应覆盖该方法以执行验证,并覆盖`isValid()`方法以返回状态。
- `setFocus()`——设置焦点至编辑器。子类可以覆盖该方法以设置焦点至编辑器内特定控件。
- `setLabelText(String)`——设置将出现于编辑器相关的标签中的文本。
- `setPreferenceName(String)`——设置正被编辑器显示的首选项的名称。
- `setPreferenceStore(IPreferenceStore)`——设置保存编辑器的值的首选项存档。
- `setPresentsDefaultValue(boolean)`——设置编辑器是否正显示默认值。
- `setPropertyChangeListener(IPropertyChangeListener)`——设置当编辑器的内容更改时,应通过`fireStateChanged()`或`fireValueChanged()`方法通知的属性更改监听器。
- `showErrorMessage(String)`——用于在首选项页的顶部显示一个错误消息的便利方法。
- `showMessage(String)`——用于在首选项页的顶部显示一个消息的便利方法。
- `store()`——保存编辑器的当前值至首选项存档。子类应覆盖`doStore()`而不是本方法。

12.2.3 PreferencePage

FieldEditorPreferencePage假设页面中的所有首选项是字段编辑器，并处理大部分关于载入、验证和保存字段编辑器内容的工作。对于更复杂的首选项页，你可以使用PreferencePage。作为替代，它是FieldEditorPreferencePage的超类。缺点是你必须自己做更多的工作。

- createContents(Composite)——为首选项页创建控件。
- doGetPreferenceStore()——返回一个页面特定的首选项存档或null以使用容器的首选项存档。子类可以在必要时覆盖本方法。
- getPreferenceStore()——为该首选项页返回首选项存档。
- isValid()——返回首选项页的内容当前是否是合法的。
- performDefaults()——使所有字段载入它们的默认值。
- performOk()——保存所有字段的值至首选项存档。
- setErrorMessage(String)——用于当字段的值不合法时在首选项页的顶部显示一个错误消息。
- setMessage(String, int)——用于在首选项页的顶部显示一个消息。
- setValid(boolean)——设置首选项页的内容当前是否是合法的。

如果你使用PreferencePage，你还可以使用不同类型的字段编辑器，但你必须自己做更多的工作（载入、验证和保存值）。额外的工作包括，当创建字段编辑器时添加一些方法。比如：

```
protected Control createContents(Composite parent) {  
    ...  
    editor = new BooleanFieldEditor(  
        "boolean", "Boolean", parent);  
    editor.setPreferencePage(this);  
    editor.setPreferenceStore(getPreferenceStore());  
    editor.load();  
    ...  
}
```

以及当用户重设值为它们的默认值：

```
protected void performDefaults() {  
    editor.loadDefault();  
    ...  
    super.performDefaults();  
}
```

以及当用户决定保存当前首选项值：

```
public boolean performOk() {  
    ...  
    editor.store();  
    return true;  
}
```

以及执行任意附加验证，而不是由字段强制执行。

12.2.4 收藏夹首选项页

对于Favorites视图中的每一列，你需要一个Boolean首选项，表示该列对于Favorites视图是否可见。首先，修改生成的PreferenceConstants类以定义首选项常量。这些常量可以由Favorites产品中的不同类共享。

```
public class PreferenceConstants
```



```

{
    public static final String
        FAVORITES_VIEW_NAME_COLUMN_VISIBLE =
            "favorites.view.name.column.visible";
    public static final String
        FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE =
            "favorites.view.location.column.visible";
}

```

然后,修改PreferenceConstants以使用布尔值首选项字段编辑器显示这两个首选项。

```

public class FavoritesPreferencePage
    extends FieldEditorPreferencePage
    implements IWorkbenchPreferencePage
{
    private BooleanFieldEditor namePrefEditor;
    private BooleanFieldEditor locationPrefEditor;

    public FavoritesPreferencePage() {
        super(GRID);
        setPreferenceStore(
            FavoritesActivator.getDefault().getPreferenceStore());
        setDescription("Favorites view column visibility:");
    }

    public void init(IWorkbench workbench) {
    }

    public void createFieldEditors() {
        namePrefEditor = new BooleanFieldEditor(
            PreferenceConstants.FAVORITES_VIEW_NAME_COLUMN_VISIBLE,
            "Show name column", getFieldEditorParent());
        addField(namePrefEditor);
        locationPrefEditor = new BooleanFieldEditor(
            PreferenceConstants.FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE,
            "Show location column", getFieldEditorParent());
        addField(locationPrefEditor);
    }
}

```

现在,当Favorites首选项页显示时,它展示了一个两列的可见性首选项(图12-5)。

12.2.5 合法性验证

首选项页总体来说是比较好的(图12-5),但存在两个问题。第一,名称和位置列的可见性默认应为true。该问题在12.3.3节和12.3.4节中讨论。第二,至少一列应一直可见。字段编辑器强制在创建过程中,基于编辑器类型和指定的参数对它们的内容进行本地合法性验证。如果你想要在不同编辑器间进行验证,那么你必须通过覆盖FieldEditorPreferencePage.checkState()方法在PreferencePage类中自己强制实现。

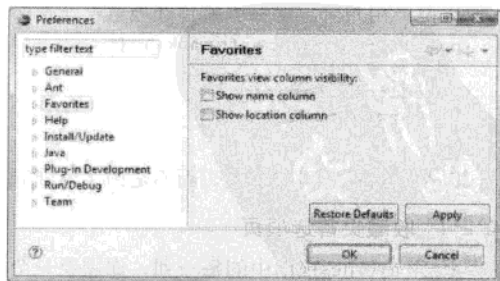


图12-5 显示列可见性的收藏夹首选项页

```
protected void checkState() {
    super.checkState();
    if (!isValid())
        return;
    if (!namePrefEditor.getBooleanValue()
        && !locationPrefEditor.getBooleanValue()) {
        setErrorMessage("Must have at least one column visible");
        setValid(false);
    }
    else {
        setErrorMessage(null);
        setValid(true);
    }
}
```

FieldEditorPropertyPage监听FieldEditor.IS_VALID属性变更事件，并随后在必要时调用checkState()和setValid()。布尔值字段编辑器从不会处于非法状态，因此它不会提交FieldEditor.IS_VALID属性更改事件，仅提交FieldEditor.VALUE属性更改事件。你必须覆盖FieldEditorPreferencePage propertyChange()方法以在收到FieldEditor.VALUE属性变更事件时调用checkState()方法。

```
public void propertyChange(PropertyChangeEvent event) {
    super.propertyChange(event);
    if (event.getProperty().equals(FieldEditor.VALUE)) {
        if (event.getSource() == namePrefEditor
            || event.getSource() == locationPrefEditor)
            checkState();
    }
}
```

现在，当两个首选项都没有被选中，在首选项页的顶部将显示一个错误消息，并且Apply和OK按钮是不可用的（图12-6）。

12.2.6 嵌套首选项页

嵌套首选项页提供一种在一个页面不够时，用于以层次结构组织相关首选项页的机制。一般地，父页面包含根级首选项或甚至只包含信息，而子首选项页关注特定的方面。

为了在Favorites产品中创建嵌套首选项页（图12-7），在插件清单中添加一个新的声明。在清单中，category属性指定了父首选项页（参见12.1节的category属性）。如果Eclipse不能找一个具有指定标识符的父页面，该首选项页将出现于根级别。

```
<page
    name="Nested Prefs"
    category="com.qualityeclipse.favorites.
prefs.view"
    class="com.qualityeclipse.favorites
.preferences.NestedPreferencePage"
    id="com.qualityeclipse.favorites.prefs.nested"/>
```

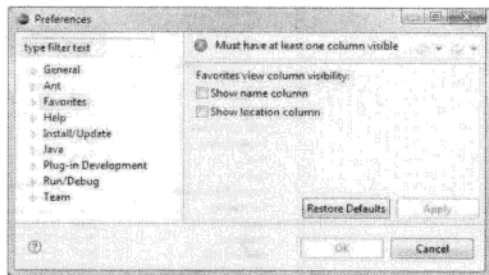


图12-6 显示错误信息的收藏夹首选项页

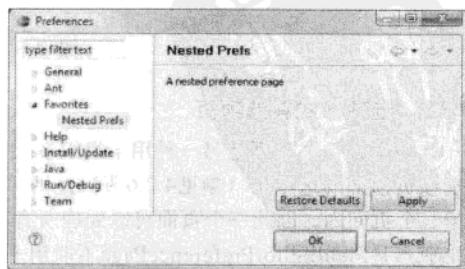


图12-7 嵌套首选项页

首选项页可以通过制定父首选项页的标识符嵌套任意层。比如,要添加一个嵌套层数为两层的收藏夹首选项页,声明可能与以下内容类似:

```
<page
  name="Nested Prefs 2"
  category="com.qualityeclipse.favorites.prefs.nested"
  class="com.qualityeclipse.favorites
    .preferences.NestedPreferencePage2"
  id="com.qualityeclipse.favorites.prefs.nested2"/>
```

提示 根首选项页可以包含关于产品的基本信息,而子首选项页包含实际的首选项。比如,图12-8中的根首选项页包含关于产品的信息,包括版本和创建日期、产品安装位置、关于创建该产品的公司的信息和生成邮件的按钮。

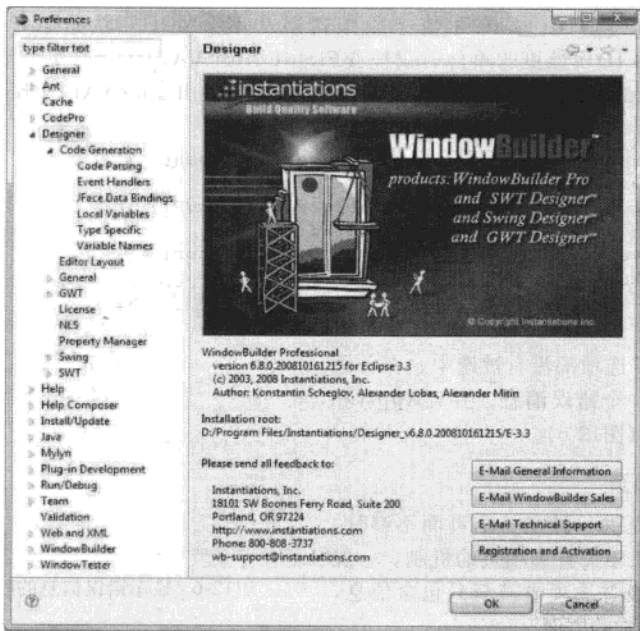


图12-8 根级首选项页

12.2.7 选项卡式首选项页

选项卡式首选项页是另一种用于组织多于一页的首选项的方法(图12-9)。在这种情况下,位于首选项页顶部的选项卡(参见4.2.6节)提供了相关首选项组之间的分隔。优点是选项卡式首选项页位于一个页面内,因此一个页面可以处理所有相互关联的字段验证。

缺点是FieldEditorPreferencePage不能用于该目的,所以你必须自己做更多的工作,将你的首选项页基于PreferencePage类作为替代(参见12.2.3节)。当然,嵌套页和选项卡式页面在需要时都可以用于同样的产品。

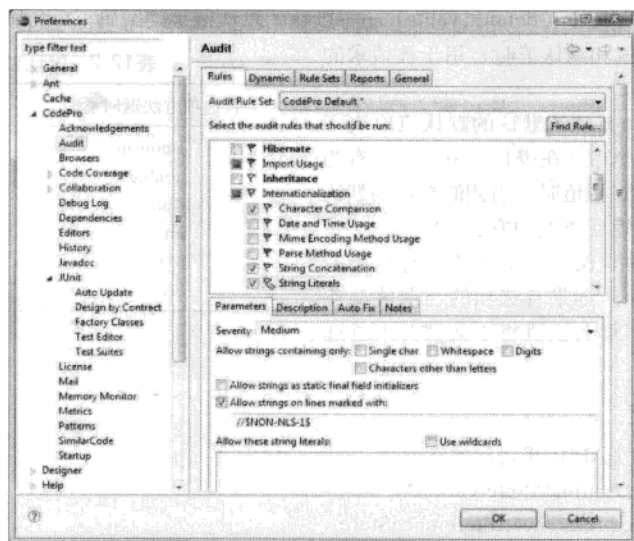


图12-9 选项卡式首选项页

12.3 首选项API

Eclipse中包括的Preference API提供了基于字符串的以平面结构存储的简单键/值对。插件底层结构为每一个插件提供了它自己的首选项存储文件，名为<plugin-id>.prefs，位于工作区元数据区域。比如，如果你使用前几节创建的Favorites首选项页更改列的可见性，那么通过查看<workspace>/metadata/.plugins/org.eclipse.core.runtime/.settings目录中的com.qualityeclipse.favorites.prefs文件你将发现：

```
#Wed Jul 30 23:55:21 EDT 2008
eclipse.preferences.version=1
favorites.view.name.column.visible=true
favorites.view.location.column.visible=true
```

提示 如果你有复杂的高度结构化的首选项数据，并且它们不仅仅是独立的键/值对，那么你可以考虑将这些首选项元素存储于一个XML格式的首选项文件中。该文件位于插件的元数据区域，与FavoritesManager存储信息的方式类似（参见7.5.2节）。

12.3.1 默认首选项

每一个首选项具有与它关联的以下三个值：

- 当前值（current value）——每一个首选项具有一个当前值。如果没有指定当前值，那么它将默认值保持一致。
- 默认值（default value）——每一个首选项具有一个默认值。如果没有指定默认值，那么它将默认的默认值一致。默认值可以通过程序指定（参见12.3.3节），也可以在插件的安装目录中的一个特殊文件中指定（参见12.3.4节）。

- 默认的首选值 (default-default value) —— 默认的首选值被硬编码至Eclipse首选项系统，并在没有指定当前值和默认值时，用于被请求的首选项。

硬编码至Eclipse系统的默认的首选值取决于用于访问该首选项的API。在表12-2中，当没有为首选项指定当前值和默认值时，右侧的默认的首选值由具有左侧所示的返回类型的方法返回。

首选项文件的内容仅表示那些值与首选项的默认值不同的首选项。如果首选项的当前值与首选项的默认值相同，那么该值将不会被写入首选项文件。

表12-2 默认首选项值

首选项方法返回类型	默认的首选值
Boolean	false
double	0.0
float	0.0f
int	0
long	0L
String	"" (空字符串)

12.3.2 访问首选项

在Eclipse中有两个API用于访问首选项：

- org.eclipse.core.runtime.Preferences
- org.eclipse.jface.preference.IPreferenceStore

你可以使用ScopedPreferenceStore的getPreferenceStore()方法（参见3.4.4节）来通过IPreferenceStore接口访问插件的首选项。如果你需要不依赖于UI插件访问首选项，那么可以通过Preferences接口使用InstanceScope、ProjectScope或ConfigurationScope（参见3.4.5节）。IPreferenceStore和Preferences接口都提供了几乎一样的访问底层首选项值的功能。这些功能可以用于访问不同格式的基于字符串的底层首选项值，包括Boolean、int和long类型等。

- getBoolean(String)——返回一个布尔类型的首选项值。true之外的其他值都被解释为false。
- getDefaultBoolean(String)——返回一个布尔类型的默认首选项值。true之外的其他值都被解释为false。
- getDefaultDouble(String)——返回一个double类型的默认首选项值。一个不表示双精度的值被解释为0.0。
- getDefaultFloat(String)——返回一个float类型的默认首选项值。一个不表示浮点类型的值被解释为0.0f。
- getDefaultInt(String)——返回一个int类型的默认首选项值。一个不表示整数类型的值被解释为0。
- getDefaultLong(String)——返回一个long类型的默认首选项值。一个不表示long类型的值被解释为0L。
- getDefaultString(String)——返回一个string类型的默认首选项值。
- getDouble(String)——返回一个double类型的首选项值。一个不表示双精度类型的值被解释为0.0。
- getFloat(String)——返回一个float类型的首选项值。一个不表示浮点类型的值被解释为0.0f。
- getInt(String)——返回一个int类型的首选项值。一个不表示整数类型的值被解释为0。
- getLong(String)——返回一个long类型的首选项值。一个不表示long类型的值被解释为0L。
- getString(String)——返回一个string类型的首选项值。
- isDefault(String)——如果指定首选项的当前值与它的默认值相同，返回true。

- `setDefault(String, boolean)`——设置指定首选项的默认值为Boolean类型。
- `setDefault(String, double)`——设置指定首选项的默认值为double类型。
- `setDefault(String, float)`——设置指定首选项的默认值为float类型。
- `setDefault(String, int)`——设置指定首选项的默认值为int类型。
- `setDefault(String, String)`——设置指定首选项的默认值为string类型。
- `setDefault(String, long)`——设置指定首选项的默认值为long类型。
- `setDefault(String)`——设置指定首选项的当前值为它的默认值。
- `setValue(String, boolean)`——设置指定首选项的值为Boolean类型。
- `setValue(String, double)`——设置指定首选项的值为double类型。
- `setValue(String, float)`——设置指定首选项的值为float类型。
- `setValue(String, int)`——设置指定首选项的值为int类型。
- `setValue(String, String)`——设置指定首选项的值为string类型。
- `setValue(String, long)`——设置指定首选项的值为long类型。

此外，还有不同的方法用于载入、保存和检查首选项对象的状态：

- `contains(String)`——返回给定首选项是否具有一个不是默认值的值或一个不是默认的默认值的默认值。
- `defaultPropertyNames()`——返回一个列表。该列表包含所有具有默认值不是默认的默认值的首选项。
- `load(InputStream)`——使用`java.util.Properties.load()`从指定的`InputStream`为首选项对象载入非默认值的首选项。默认的首选项值不受影响。
- `needsSaving()`——返回是否有至少一个首选项值在上一次保存首选项后已经更改（参见`store()`）。
- `propertyNames()`——返回一个列表。该列表包含所有具有当前值不是它们的默认值的首选项的名称。
- `store(OutputStream, String)`——使用`Properties.store()`保存非默认值的首选项值至指定的`OutputStream`，并重设dirty标志位。因此`needsSaving()`直到修改了一个首选项值之前都将返回false。

注意 根据3.4.6节中所描述的，如果你的插件类是`org.eclipse.core.runtime.Plugin`而不是`org.eclipse.ui.plugin.AbstractUIPlugin`的子类，或者你使用一种不是调用`getPreferenceStore()`的方法访问首选项，那么你必须修改`stop()`以保存你的首选项，以使它们可以在会话间得到保持。

12.3.3 在程序中指定默认值

默认值可以在第一次启动插件时使用Preferences API在程序中指定。扩展你的插件的首选项初始化类的`initializeDefaultPreferences()`方法并调用不同的`setDefault*`方法。对于Favorites产品而言，修改本章较早时候创建的`PreferenceInitializer`类并实现`initializeDefaultPreferences()`方法为名称列可见性首选项设置默认值。现在，当第一次显示Favorites首选项页时，Show name column首选项将已经被选中。

```
public class PreferenceInitializer
    extends AbstractPreferenceInitializer {
    public void initializeDefaultPreferences() {
```

```

    IPreferenceStore store = FavoritesActivator.getDefault()
        .getPreferenceStore();
    store.setDefault(
        PreferenceConstants.FAVORITES_VIEW_NAME_COLUMN_VISIBLE,true);
}
}

```

PreferenceInitializer类被org.eclipse.core.runtime.preferences扩展点在plugin.xml文件中引用。该扩展点自动被稍早使用的Preference Page模板所扩展。

```

<extension point="org.eclipse.core.runtime.preferences">
    <initializer class="com.qualityeclipse.favorites.
        preferences.PreferenceInitializer"/>
</extension>

```

一旦你已经在程序中为名称列可见性首选项指定了true为默认值，它将出现于com.qualityeclipse.favorites.prefs文件（参见12.3节）的唯一时机是当该首选项不是ture时。

12.3.4 在文件中指定默认值

默认首选项也可以在一个特殊的preferences.ini文件中指定。该文件位于插件的安装目录。该文件具有与com.qualityeclipse.favorites.prefs文件同样的格式，并可以在安装插件时安装。将默认值放置在文件中的优点是它从代码释放它们，使得它们在不需修改代码的情况下更容易被修改。缺点是以这种方式指定的默认值不能像它们在程序中被指定的那样进行动态调整。然而，一般地，一个默认的首选项规格不需要这样的灵活性。对于Favorites产品来说，你将在项目根下添加一个新的preferences.ini文件。该文件包含一行内容，为位置列可见性指定了默认值：

```
favorites.view.location.column.visible=true
```

提示 你可以使用工作台首选项对话框中的General > Editors > File Associations页（参见1.3.1节）将内部文本编辑器关联所有的*.ini文件，以使得双击preferences.ini文件将在Eclipse内部为该文件打开一个文本编辑器。

为了完成该过程，必须修改构建脚本以包含该新的preferences.ini文件作为产品的一部分（参见第19章以了解更多关于构建产品的内容）。一旦你已经指定true作为位置列可见性首选项的默认值，它将在com.qualityeclipse.favorites.prefs文件（参见12.3节）中出现的唯一时机是当首选项不是true时。

12.3.5 关联收藏夹视图

既然已经完成Favorites首选项页，你可关联这些首选项至Favorites视图。首先，使用Extract Constant重构放置初始列宽度至常量中：

```

private static final int NAME_COLUMN_INITIAL_WIDTH = 200;
private static final int LOCATION_COLUMN_INITIAL_WIDTH = 450;

```

然后，你将创建一个新的updateColumnWidths()方法。它在创建完表之后从createPartControl (Composite)方法中调用。

```

private void updateColumnWidths() {
    IPreferenceStore prefs = FavoritesActivator
        .getDefault().getPreferenceStore();

    boolean showNameColumn = prefs.getBoolean(
        PreferenceConstants.FAVORITES_VIEW_NAME_COLUMN_VISIBLE);
}

```

```
nameColumn.setWidth(  
    showNameColumn ? NAME_COLUMN_INITIAL_WIDTH : 0);  
  
boolean showLocationColumn = prefs.getBoolean(  
    PreferenceConstants.FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE);  
locationColumn.setWidth(  
    showLocationColumn ? LOCATION_COLUMN_INITIAL_WIDTH : 0);  
}
```

当完成这两个更改后，Favorites视图将根据Favorites首选项页中指定地显示名称和位置列。

12.3.6 监听首选项更改

当第一次打开Favorites视图时，列遵循Favorites首选项页中指定的设置，但如果当Favorites视图已经打开时首选项被更改将会发生什么？为了使Favorites视图与Favorites首选项页中指定的首选项保持同步，你需要添加监听器至包含首选项的对象。回到FavoritesView，你可以添加一个新的propertyChangeListener字段。该字段监听属性更改事件，并在合适时调用updateColumnWidths()。

```
private final IPropertyChangeListener propertyChangeListener  
    = new IPropertyChangeListener() {  
    public void propertyChange(PropertyChangeEvent event) {  
        if (event.getProperty().equals(  
            FAVORITES_VIEW_NAME_COLUMN_VISIBLE_PREF)  
            || event.getProperty().equals(  
            FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE_PREF))  
            updateColumnWidths();  
    }  
};
```

该新的propertyChangeListener必须当在createPartControl()方法的末尾创建视图时，作为一个监听器添加。

```
FavoritesActivator.getDefault().getPreferenceStore()  
    .addPropertyChangeListener(propertyChangeListener);
```

监听器必须在视图关闭时在dispose()方法中移除。

```
FavoritesActivator.getDefault().getPreferenceStore()  
    .removePropertyChangeListener(propertyChangeListener);
```

12.4 RFRS相关事项

《RFRS Requirements》的“用户界面”一节包含了一个与首选项相关的要求。它来源于Eclipse UI准则。

首选项对话框使用 (RFRS 3.5.25)

用户界面准则#15.1是一个要求。它说明：

全局选项应在首选项对话框中显示。

当你需要向用户显示全局选项时，必须创建一个新的首选项页。比如，Java编译的全局首选项在Preferences对话框作为一组首选项页显示。如果这些首选项被更改，它们将影响整个Java插件。

为了通过该测试，展示你产品的首选项页的一个样本并说明首选项设置在它的内部控制全局选项。更改一个首选项，然后重启Eclipse以显示首选项的值被恰当的保持。对于Favorites首选项，你应展示列可见性选项全局上影响在所有透视图中打开的所有Favorites视图中显示的列。

12.5 总结

几乎所有重要的插件都将包含控制它的执行和与用户的交互的全局选项。本章探讨了Eclipse首选项页API并讨论了开发者可用的选择。这些选择可以用于创建简单和复杂的首选项页。它还说明了如何在工作区会话间保持首选项设置。

参考文献

本书资源(2.9节).

Creasey, Tod, "Preferences in the Eclipse Workbench UI," August 15, 2002 (www.eclipse.org/articles/Article-Preferences/preferences.htm).

Cooper, Ryan, "Simplifying Preference Pages with Field Editors," August 21, 2002 (www.eclipse.org/articles/Article-Field-Editors/field_editors.html).



第13章 属 性

首选项适用于插件或功能块，而属性适用于资源或其他出现于Eclipse环境的对象。访问对象属性的一种典型方法是从它的上下文菜单选择Properties命令，打开Properties对话框。另一种方法是打开Properties视图。它显示被选中对象的属性。

本章包含在特定对象上创建属性，和在对象的Properties对话框和Properties视图中显示该属性的内容。

13.1 创建属性

你需要为颜色添加属性，并添加注释至Favorites产品。Color属性将决定在Favorites视图中显示项所使用的颜色，而comment属性将被显示为项的悬停帮助。

由于属性与对象关联，你必须决定哪种类型的对象将包含属性。Color属性将会被添加至Favorites项。当从Favorites视图中移除项时，Color属性将会被忽略。相反地，你需要将comment属性关联至Favorites项后台的资源，这样当资源被移除然后添加至Favorites视图时，comment属性将会被保留。

13.1.1 FavoriteItem属性

属性可以以多种不同的方式关联至对象，但一般地，属性值是通过对象自身的get和set方法访问。对于Favorites项来说，访问方法需要为新的Color属性添加至IFavoriteItem接口：

```
Color getColor();  
void setColor(Color color);
```

因为该属性在所有Favorites项中都相同地实现，所以你需要将这种方法放置于一个名为BasicFavoriteItem的抽象超类。它将由你所有的其他项类型扩展。

```
public class BasicFavoriteItem  
{  
    private Color color;  
    private static Color defaultColor;  
  
    public Color getColor() {  
        if (color == null)  
            return getDefaultColor();  
        return color;  
    }  
    public void setColor(Color color) {  
        this.color = color;  
        FavoritesManager.getManager().fireFavoritesItemChanged(this);  
    }  
    public static Color getDefaultColor() {  
        if (defaultColor == null)  
            defaultColor = getColor(new RGB(0, 0, 0));  
        return defaultColor;  
    }  
}
```



```

    }
    public static void setDefaultColor(Color color) {
        defaultColor = color;
    }
}

```

颜色更改必须向所有感兴趣的监听器广播。添加以下代码至FavoritesManager类:

```

private Collection<IFavoritesListener> itemListeners =
    new ArrayList<IFavoritesListener>();

public void addFavoritesListener(IFavoritesListener listener) {
    itemListeners.add(listener);
}
public void removeFavoritesListener(IFavoritesListener listener) {
    itemListeners.remove(listener);
}
public void fireFavoritesItemChanged(BasicFavoriteItem item) {
    for (Iterator<IFavoritesListener> iterator =
        itemListeners.iterator(); iterator.hasNext();)
        iterator.next().favoritesItemChanged(item);
}

```

属性的类型分两种:持久(persistent)属性和会话(session)属性。持久属性在多个工作台会话之间将得到保存,而会话属性值在Eclipse退出时将被丢弃。要在多个会话间保存Color属性,你将需要修改7.5.2节中列出的载入和保存方法。该内容将留给读者作为练习。到目前为止,Color属性将不会在会话间得到保存。

Color对象具有一个底层操作系统资源,并且必须被恰当管理。添加BasicFavoriteItem实用方法以缓存、重用和释放颜色:

```

private static final Map<RGB, Color> colorCache
    = new HashMap<RGB, Color>();

public static Color getColor(RGB rgb) {
    Color color = colorCache.get(rgb);
    if (color == null) {
        Display display = Display.getCurrent();
        color = new Color(display, rgb);
        colorCache.put(rgb, color);
    }
    return color;
}

public static void disposeColors() {
    Iterator<Color> iter = colorCache.values().iterator();
    while (iter.hasNext())
        iter.next().dispose();
    colorCache.clear();
}

```

当Favorites插件关闭时,你必须清理所有你管理的Color对象。添加以下代码至FavoritesActivator.stop()方法:

```

BasicFavoriteItem.disposeColors();

```

13.1.2 资源属性

Eclipse具有一种通用机制用于关联属性和资源。你可以使用这些资源存储资源注释。IResource中的方法提供了会话属性和持久属性。这些会话属性将在Eclipse退出时被忽略，而这些持久属性将在多个工作区会话间保存。两种类型的属性都可以用于决定一个操作是否应是可见的（参见6.7.2节）。

- `getPersistentProperty(QualifiedName)`——返回由给定键标识的资源的持久属性的值。如果该资源没有该属性，则返回null。这些属性在不同会话间将保存。
- `getSessionProperty(QualifiedName)`——返回由给定键标识的资源的会话属性的值。如果该资源没有该属性，则返回null。这些属性将在Eclipse退出时被忽略。
- `setPersistentProperty(QualifiedName, String)`——设置由给定键标识的资源的持久属性的值。如果提供的值是null，该持久属性将从资源中移除。这些属性在不同会话间保留。
- `setSessionProperty(QualifiedName, Object)`——设置由给定键标识的资源的会话属性的值。如果提供的值是null，该会话属性将从资源中移除。
- 这些方法中的`QualifiedName`参数用于存储并获取属性值的键。依据常规，一个键由插件标识符和一个在插件内标识属性的字符串组成。对于Favorites产品来说，在`BasicFavoriteItem`类中为`comment`属性定义一个常量键：

```
public static final QualifiedName COMMENT_PROPKEY =  
    new QualifiedName(FavoritesActivator.PLUGIN_ID, "comment");
```

如同稍早讨论的那样，有两种类型的属性：持久属性和会话属性。持久属性在多个工作台会话间保留，而会话属性将在Eclipse退出时被忽略。你需要`comment`属性以在多个工作台会话间保留，所以像如下所示使用`getPersistentProperty()`和`setPersistentProperty()`方法：

```
String comment =  
    resource.getPersistentProperty(  
        BasicFavoriteItem.COMMENT_PROPKEY);  
resource.setPersistentProperty(  
    BasicFavoriteItem.COMMENT_PROPKEY,  
    comment);
```

如果资源对象不具有一个与其关联的Favorites注释，那么你需要显示一个默认注释。添加`BasicFavoriteItem`方法以访问默认注释。

```
public static final String COMMENT_PREFKEY = "defaultComment";  
  
public static String getDefaultComment() {  
    return FavoritesActivator.getDefault().getPreferenceStore()  
        .getString(COMMENT_PREFKEY);  
}  
  
public static void setDefaultComment(String comment) {  
    FavoritesActivator.getDefault().getPreferenceStore()  
        .setValue(COMMENT_PREFKEY, comment);  
}
```

13.2 在属性对话框中显示属性

当你定义了属性后，你需要在Properties对话框中显示和编辑这些属性。首先，添加一个页面至已有的资源Properties对话框以显示和编辑`comment`属性。然后，为Favorites视图中的选中Favorites项打开一个Properties对话框以显示和编辑`Color`和`comment`属性。

13.2.1 声明属性页

为了创建资源Properties对话框中新的Property页，你需要在Favorites插件清单中声明该页面。该声明引用新的FavoriteResourcePropertyPage类，该类处理创建和新页面中的用户交互（图13-1）。

为了在插件清单中创建属性页声明，编辑Favorites的plugin.xml，切换至Extensions页，在New Extensions对话框中点击Add...，选择org.eclipse.ui.propertyPages并点击Finish。



图13-1 属性页声明

为了创建新页面，右键点击插件清单编辑器Extensions页面的org.eclipse.ui.propertyPages，并选择New > page。点击新建的com.qualityeclipse.favorites.page1页面声明以编辑它的属性，并输入以下属性。

- id——“com.qualityeclipse.favorites.resourcePropertyPage”
用于标识Property页的唯一名称。
- name——“Favorite Properties”
Property页的可读名称。
- class——“com.qualityeclipse.favorites.properties.FavoriteResourcePropertyPage”

实现org.eclipse.ui.IWorkbenchPropertyPage的类的完全合格名称。点击类属性值左侧的class按钮以自动生成FavoriteResourcePropertyPage类。

- icon——“icons/sample.gif”

图标的路径。该图标与名称属性一起显示于UI中。使用图标属性值右侧的Browse...按钮以选择Favorites项目中的sample.gif文件。

- objectClass——留为空白（不建议的）

注册至页面的类的完全合格名称。该方法是不建议的。使用下面描述的enabledWhen表达式。

- nameFilter——留为空白

一个可选属性。它允许有条件地注册一个应用于目标对象名称的通配符匹配。你不需要为收藏夹属性使用过滤器，但如果你想要限制它仅适用于Java源文件，输入“*.java”。

- adaptable——留为空白（不建议的）

表示匹配于IResource的类型是否应使用Property页。该方法是不建议的。使用下面描述的enabledWhen表达式。

- category——留为空白

表示属性树中页面的位置的路径。路径可以是一个父节点ID或一个由“/”分隔的ID序列。它表示从根节点开始的完全路径。

添加一个enabledWhen表达式至上面的声明以指定FavoritesResourcePropertyPage应在何时可见。这些表达式具有与命令表达式同样的格式和表达能力（参见6.2.10节）。当完成时，声明应与以下内容类似：

```
<page class="com.qualityeclipse.favorites.properties
        .FavoriteResourcePropertyPage"
      icon="icons/sample.gif"
      id="com.qualityeclipse.favorites.resourcePropertyPage"
      name="Favorite Properties">
  <enabledWhen>
    <adapt type="org.eclipse.core.resources.IResource"/>
  </enabledWhen>
</page>
```

上面的声明使得FavoritesResourcePropertyPage对于所有使用IAdaptable接口，适配于IResource接口的对象都是可见的。我们需要为IFavoriteItem的实例特别添加一个新的属性页。但由于IFavoriteItem它自身是与IResource匹配的，新建的属性页和FavoritesResourcePropertyPage页都将出现于Property对话框中。要防止该问题的发生，修改上面的声明以排除的IFavoriteItem实例。

```
<page class="com.qualityeclipse.favorites.properties
        .FavoriteResourcePropertyPage"
      icon="icons/sample.gif"
      id="com.qualityeclipse.favorites.resourcePropertyPage"
      name="Favorite Properties">
  <enabledWhen>
    <and>
      <adapt type="org.eclipse.core.resources.IResource"/>
      <not>
        <instanceof value=
          "com.qualityeclipse.favorites.model.IFavoriteItem"/>
      </not>
    </and>
  </enabledWhen>
</page>
```

添加另一个与上面的声明类似的Property页声明以将FavoriteItemPropertyPage与IFavoriteItem的

实例关联起来。该新页面是对包含一个附加属性字段的FavoriteResourcePropertyPage页的改进。当完成时, 声明应与以下内容看起来类似:

```
<page class="com.qualityeclipse.favorites.properties
    .FavoriteItemPropertyPage"
    icon="icons/sample.gif"
    id="com.qualityeclipse.favorites.favoritesItemPropertyPage"
    name="Favorite Properties">
    <enabledWhen>
    <instanceof value=
        "com.qualityeclipse.favorites.model.IFavoriteItem"/>
    </enabledWhen>
</page>
```

提示 大部分的Property页不具有一个相关的图标。如果你将一个图标关联至你的Property页, 那么Property页的列表将由于所有其他Property页的名称的前面的空白位置看起来十分有趣。为了说明这一点, 一个图标被关联至收藏夹Property页 (图13-2), 但我们建议你不要这样做。

一种将资源局限于属性页适用的资源的方法是按照刚描述的添加nameFilter属性。另一种方法是通过右键点击Favorites Property声明添加一个过滤器子元素。该属性页声明位于插件清单编辑器的Extensions页面。然后选择New > filter。filter子元素指定了一个属性名称和值。

- name——对象属性的名称。
- value——对象属性的值。与name属性一起, 名/值对用于为属性页定义目标对象。

显示属性的选中对象必须在Property页显示之前具有该属性的指定值。比如, 要为只读文件显示一个Property页, 你需要指定一个filter子元素。该子元素必须name="readOnly"和value="true"。为了使用过滤器子元素, 选中对象必须使用org.eclipse.ui.IActionFilter接口。Eclipse工作台资源类型, 如IFile和IFolder, 当前实现了该接口。

13.2.2 创建资源属性页

当完成了Property页声明后, 你需要填充由New Java Class向导生成的FavoriteResourcePropertyPage类存根 (class stub)。我们从填充一些字段和createContents()方法开始。由于FavoriteResourcePropertyPage扩展了PropertyPage并从首选项页框架继承了行为 (参见12.2.3节), createContents()方法将被调用以用于创建和初始化页面控件 (图13-2)。

```
private Text textField;

protected Control createContents(Composite parent) {
    Composite panel = new Composite(parent, SWT.NONE);
    GridLayout layout = new GridLayout();
    layout.marginHeight = 0;
    layout.marginWidth = 0;
    panel.setLayout(layout);

    Label label = new Label(panel, SWT.NONE);
    label.setLayoutData(new GridData());
    label.setText(
        "Comment that appears as hover help in the Favorites view:");
    textField = new Text(panel, SWT.BORDER | SWT.MULTI | SWT.WRAP);
    textField.setLayoutData(new GridData(GridData.FILL_BOTH));
    textField.setText(getCommentPropertyValue());
}
```

```
    return panel;
}
```

PropertyPage类包含一个getElement()访问方法用于获取当前属性正被编辑的对象。为获取与设置当前元素的相关注释创建访问方法:

```
protected String getCommentPropertyValue() {
    IResource resource =
        (IResource) getElement().getAdapter(IResource.class);
    try {
        String value =
            resource.getPersistentProperty(
                BasicFavoriteItem.COMMENT_PROPKEY);
        if (value == null)
            return BasicFavoriteItem.getDefaultComment();
        return value;
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return e.getMessage();
    }
}

protected void setCommentPropertyValue(String comment) {
    IResource resource =
        (IResource) getElement().getAdapter(IResource.class);
    String value = comment;
    if (value.equals(BasicFavoriteItem.getDefaultComment()))
        value = null;
    try {
        resource.setPersistentProperty(
            BasicFavoriteItem.COMMENT_PROPKEY,
            value);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
}
```

由于FavoriteResourcePropertyPage扩展了PropertyPage并从Preference页框架继承了行为(参见12.2.3节),当点击OK按钮时将调用performOk()方法,给予属性页一个存储值的机会。

```
public boolean performOk() {
    setCommentPropertyValue(textField.getText());
    return super.performOk();
}
```

当完成了这些工作后,为Favorites项目打开Properties对话框将显示Favorites Property页(图13-2)。

13.2.3 创建收藏夹项资源页

当我们已经成功添加了一个Property页至资源Properties对话框后,你现在需要为IFavoriteItem的实例显示一个类似的Property页。但它包含一个附加字段。然而,上一节描述的资源属性页仅显示一个comment属性,该新的FavoriteItemPropertyPage将扩展FavoriteResourcePropertyPage页以添加一个字段用于显示Color属性。我们从创建一个新的类并添加createContents()方法开始。

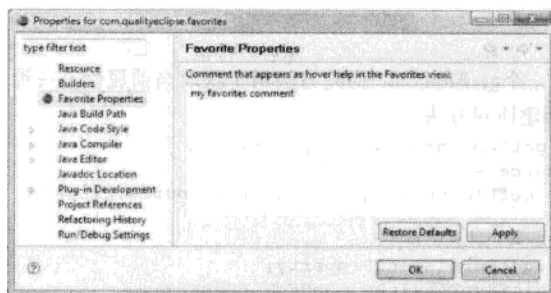


图13-2 收藏夹项目的收藏夹资源属性页

```
private ColorSelector colorSelector;

protected Control createContents(Composite parent) {
    Composite panel = new Composite(parent, SWT.NONE);
    GridLayout layout = new GridLayout();
    layout.numColumns = 2;
    layout.marginHeight = 0;
    layout.marginWidth = 0;
    panel.setLayout(layout);

    Label label = new Label(panel, SWT.NONE);
    label.setLayoutData(new GridData());
    label.setText("Color of item in Favorites View:");

    colorSelector = new ColorSelector(panel);
    colorSelector.setColorValue(getColorPropertyValue());
    colorSelector.getButton().setLayoutData(
        new GridData(100, SWT.DEFAULT));

    Composite subpanel = (Composite) super.createContents(panel);
    GridData gridData = new GridData(GridData.FILL_BOTH);
    gridData.horizontalSpan = 2;
    subpanel.setLayoutData(gridData);

    return panel;
}
```

创建访问方法用于获取和设置选中Favorites项的颜色。

```
protected RGB getColorPropertyValue() {
    IFavoriteItem item = (IFavoriteItem) getElement();
    Color color = item.getColor();
    return color.getRGB();
}

protected void setColorPropertyValue(RGB rgb) {
    IFavoriteItem item = (IFavoriteItem) getElement();
    Color color = BasicFavoriteItem.getColor(rgb);
    if (color.equals(BasicFavoriteItem.getDefaultColor()))
        color = null;
    item.setColor(color);
}
```

创建一个performOk()方法将颜色值存储回选中Favorites项:

```
public boolean performOk() {
    setColorPropertyValue(colorSelector.getColorValue());
    return super.performOk();
}
```

13.2.4 打开属性对话框

你已经创建了一个新的、经过改进的FavoriteItemPropertyPage以显示Favorites项属性,但该页将仅出现于IFavoriteItem实例所打开的Properties对话框中。为了在一个IFavoriteItem的实例上打开Properties对话框,你需要添加一个Properties命令至Favorites视图上下文菜单的末尾。

org.eclipse.ui插件已经提供了一个命令用于打开Properties对话框,所以声明以下新的菜单项添加项(参见6.2.6节)以添加org.eclipse.ui.file.properties命令至Favorites视图上下文菜单:

```
<menuContribution locationURI=
    "popup:com.qualiteclipse.favorites.views.FavoritesView
    ?after=other">
    <command commandId="org.eclipse.ui.file.properties" />
</menuContribution>
```

我们需要该新的命令出现于Favorites视图上下文菜单末尾的“其他”组中,所以添加以下行至FavoritesView.fillContextMenu()方法的末尾:

```
menuMgr.add(new Separator("other"));
```

现在,在Favorites视图上下文菜单中选择Properties将为选中的Favorites项显示Properties对话框(图13-3)。

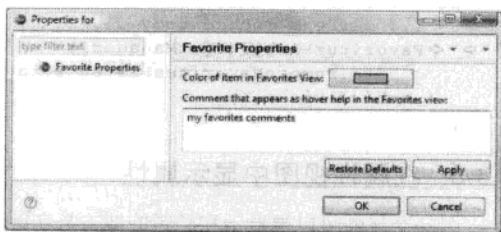


图13-3 收藏夹项目的收藏夹项属性页

13.2.5 IColorProvider

一旦我们可以使用Properties对话框指定颜色,我们必须修改Favorites视图标签提供者(参见7.2.5节)以使收藏夹项以指定颜色显示。修改标签提供者以实现IColorProvider接口并添加以下方法:

```
public Color getForeground(Object element) {
    if (element instanceof IFavoriteItem)
        return ((IFavoriteItem) element).getColor();
    return null;
}
```

```
public Color getBackground(Object element) {
    return null;
}
```

此外,第一列的标签提供者必须修改以显示收藏夹项的颜色:

```
private void createInlineEditor() {
    TableViewerColumn column =
        new TableViewerColumn(viewer, nameColumn);
    column.setLabelProvider(new ColumnLabelProvider() {
        public String getText(Object element) {
            return ((IFavoriteItem) element).getName();
        }
        public Color getForeground(Object element) {
            return ((IFavoriteItem) element).getColor();
        }
    });
}
```

```

    }
    });
    ... existing method body ...
}

```

当完成了上面的标签提供者的修改后，一旦你使用Properties对话框指定了颜色，你必须关闭并重新打开Favorites视图以查看颜色更改。要完成该任务以使颜色更改立即在Favorites视图中显示，修改FavoritesView以监听由FavoritesManager发出的颜色更改广播。如下所示：

```

private final IFavoritesListener favoritesItemListener =
    new IFavoritesListener() {
        public void favoritesItemChanged(IFavoriteItem item) {
            viewer.update(item, null);
        }
    };

private void createTableViewer(Composite parent) {
    ... current method body ...
    FavoritesManager.getManager()
        .addFavoritesListener(favoritesItemListener);
}

public void dispose() {
    ... current method body ...
    FavoritesManager.getManager()
        .removeFavoritesListener(favoritesItemListener);
    super.dispose();
}

```

13.3 在属性视图中显示属性

另一个属性可以被显示和编辑的地方是在Properties视图中。Properties视图检查工作台选择以决定是否选中对象支持org.eclipse.ui.views.properties.IPropertySource接口。对象可以通过两种方法支持IPropertySource接口。一种是直接实现IPropertySource接口，另一种是实现getAdapter()方法以返回一个实现IPropertySource接口的对象（图13-4）。

13.3.1 属性视图API

IPropertySource接口为每一个将要在Properties视图中显示的属性提供了一个描述符，和用于获取和设置属性值的方法。在方法中的随后的id参数是该属性的描述符的相关标识符。

- `getPropertyDescriptors()`——返回一个描述符数组，每一个描述符对应于将要在Properties视图中显示的一个属性。
- `getPropertyValue(Object)`——返回具有指定标识符的属性的值。
- `isPropertySet(Object)`——如果由标识符指定的属性具有一个与它的默认值不同的值，则返回true。
- `resetPropertyValue(Object)`——设置由标识符指定的属性的值为它的默认值。
- `setPropertyValue(Object, Object)`——设置由标识符指定的属性的值为指定值。

可选地，对象可以实现IPropertySource2接口以允许一个属性的更简单的表示。该属性具有一个默认值并可以被重设。

- `isPropertyResettable(Object)`——返回具有指定id的属性的值是否可以被重设为默认值。

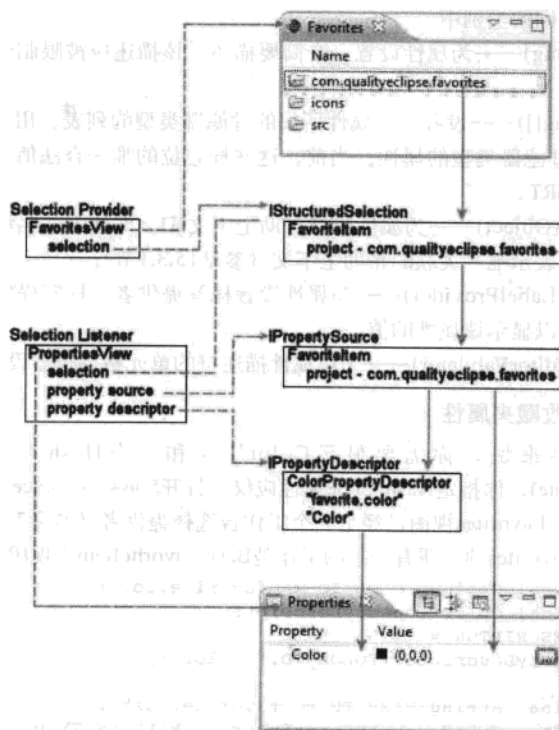


图13-4 从选择到属性视图

- `isPropertySet(Object)`——与 `IPropertySource` 中的对应方法十分类似。区别在于如果一个对象实现了 `IPropertySource2`，那么如果引用的属性不具有一个有意义的默认值，该方法应当返回 `true` 而不是 `false`。

属性描述符（实现 `IPropertyDescriptor` 接口的对象）包含一个属性标识符并在需要时为 `Properties` 视图创建一个属性编辑器。Eclipse 提供了一些 `IPropertyDescriptor` 接口的实现（图13-5）。

`PropertyDescriptor` 的实例是使用一个属性标识符和属性的显示名称进行创建的。如果一个对象有许多属性，那么通过在组中每一个描述符上调用 `setCategory()` 将相似属性分组显示是很有用的。

其他有用的方法包括：

- `setAlwaysIncompatible(boolean)`——设置一个标志位，表示属性描述符是否总是与其他所有属性描述符不兼容的。设置该标志位将阻止一个属性在多个选择中显示。
- `setCategory(String)`——设置属性所属的类别的名称。属于相同类别的属性将被分至相同组显示。该本地字符串将向用户显示。如果类别没有设置任何的描述符，则属性将出现于属性视图的最高级而不分组显示。如果类别设置至少一个描述符，那么任意具有一个未设置类别的

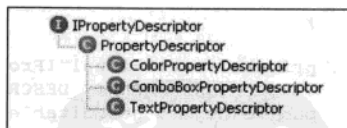


图13-5 IPropertyDescriptor 层次结构

描述符将出现于不同的类别中。

- `setDescription(String)`——为属性设置一个简要描述。该描述应被限制于一行内，以使它可以在状态栏中显示。
- `setFilterFlags(String[])`——设置一个属性所属的过滤器类型的列表。用户可以切换过滤器以显示/隐藏属于一个过滤器类型的属性。当前，这些标志位的唯一合法值是 `IPROPERTY_SHEET_ENTRY_FILTER_ID_EXPERT`。
- `setHelpContextIds(Object)`——为属性设置帮助上下文ID。即使方法名称是复数形式，仅可以指定一个字符串，表示唯一关联的帮助上下文（参见15.3.1节）。
- `setLabelProvider(ILabelProvider)`——为属性设置标签提供者。标签提供者用于获取文本（和可能已有的图像）以显示该属性的值。
- `setValidator(ICellEditorValidator)`——为该属性描述符的单元格编辑器设置输入验证器。

13.3.2 属性视图中的收藏夹属性

对于 Favorites 产品来说，你需要显示 Color 属性和一个 Hash Color 属性。通过使用 `setAlwaysIncompatible(true)`，你指定 Hash Code 属性应仅当打开 Show Advanced Properties 选项时，出现于 Properties 对话框中。Favorites 视图已经是一个工作台选择提供者（参见7.4.1节），因此 Properties 视图已经在检查选中的 Favorites 项。所有剩下的工作是 `BasicFavoriteItem` 实现 `IPROPERTY_SOURCE_2` 接口。

```
private static final String COLOR_ID = "favorite.color";
private static final ColorPropertyDescriptor
COLOR_PROPERTY_DESCRIPTOR =
    new ColorPropertyDescriptor(COLOR_ID, "Color");

private static final String HASH_ID = "favorite.hash";
private static final TextPropertyDescriptor HASH_PROPERTY_DESCRIPTOR
    = new TextPropertyDescriptor(HASH_ID, "Hash Code");
static {
    HASH_PROPERTY_DESCRIPTOR.setCategory("Other");
    HASH_PROPERTY_DESCRIPTOR.setFilterFlags(
        new String[] { IPROPERTY_SHEET_ENTRY_FILTER_ID_EXPERT });
    HASH_PROPERTY_DESCRIPTOR.setAlwaysIncompatible(true);
}

private static final IPROPERTY_DESCRIPTOR[] DESCRIPTORS =
    { COLOR_PROPERTY_DESCRIPTOR, HASH_PROPERTY_DESCRIPTOR };
public Object getEditableValue() {
    return this;
}

public IPROPERTY_DESCRIPTOR[] getPropertyDescriptors() {
    return DESCRIPTORS;
}

public Object getPropertyValue(Object id) {
    if (COLOR_ID.equals(id)) {
        return getColor().getRGB();
    }
    if (HASH_ID.equals(id)) {
        return new Integer(hashCode());
    }
    return null;
}
```

```

public boolean isPropertyResettable(Object id) {
    if (COLOR_ID.equals(id))
        return true;
    return false;
}

public boolean isPropertySet(Object id) {
    if (COLOR_ID.equals(id))
        return getColor() != getDefaultColor();
    if (HASH_ID.equals(id)) {
        // Return true for indicating that hash
        // does not have a meaningful default value.
        return true;
    }
    return false;
}

public void resetPropertyValue(Object id) {
    if (COLOR_ID.equals(id))
        setColor(null);
}

public void setPropertyValue(Object id, Object value) {
    if (COLOR_ID.equals(id))
        setColor(getColor((RGB) value));
}

```

现在，在Favorites视图中选择一项时，属性视图将为该项显示Color属性。当打开Show Advanced Properties选项时，Hash Code属性将显示（图13-6）。

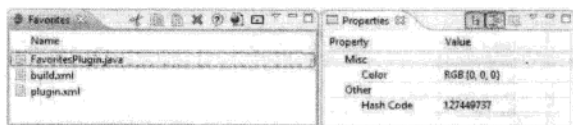


图13-6 属性视图显示高级属性

13.4 属性页作为首选项页重用

由于PropertyPage从PreferencePage继承，通过一点点工作你就可以重用一個Property页为一个Preference页。在这种情况下，你需要重用FavoriteItemPropertyPage为一个Preference页以指定Color和comment属性的默认值。为了完成该任务，创建一个新的FavoriteDefaultsPreferencePage作为FavoriteItemPropertyPage的子类。它实现org.eclipse.ui.IWorkbenchPreferencePage并覆盖属性访问方法。

```

public class FavoriteDefaultsPreferencePage
    extends FavoriteItemPropertyPage
    implements IWorkbenchPreferencePage
{
    public void init(IWorkbench workbench) {
    }

    protected RGB getColorPropertyValue() {

```

```

        return BasicFavoriteItem.getDefaultColor().getRGB();
    }

    protected void setColorPropertyValue(RGB rgb) {
        BasicFavoriteItem.setDefaultColor(
            BasicFavoriteItem.getColor(rgb));
    }

    protected String getCommentPropertyValue() {
        return BasicFavoriteItem.getDefaultComment();
    }

    protected void setCommentPropertyValue(String comment) {
        BasicFavoriteItem.setDefaultComment(comment);
    }
}

```

然后，在Favorites插件清单中创建一个具有以下属性的Preference页声明（参见12.1节）：

- category = “com.qualityeclipse.favorites.prefs.view”
- class = “com.qualityeclipse.favorites.properties.FavoriteDefaultsPreferencePage”
- id = “com.qualityeclipse.favorites.prefs.defaults”
- name = “Defaults”

完成以上工作后，Defaults首选项页作为Favorites首选项页（图13-7）的一个子页面，出现于工作台Preferences对话框中。

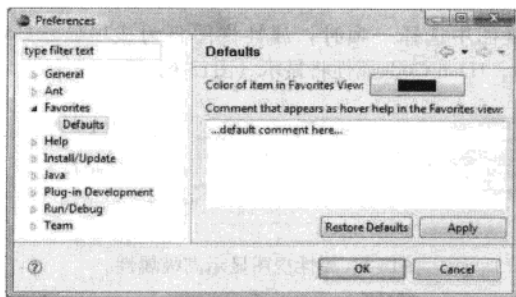


图13-7 收藏夹默认首选项页

13.5 RFRS相关事项

《RFRS Requirements》的“用户界面”一节包含了一个关于属性的要求。它来源于Eclipse UI 准则。

用于快速访问的属性视图（RFRS 3.5.21）

用户界面准则#10.1是一个要求。它说明：

使用属性视图在快速访问十分重要时，以编辑对象的属性，并且你将在对象间快速切换。

为了通过这项测试，展示你插件中的对象具有可以使用Properties视图进行编辑的属性。对于Favorites视图，你需要展示每一个Favorites项在Properties视图内显示它的颜色和hash码（图13-6）。

13.6 总结

许多插件将需要创建和管理它们自己的插件特定的资源。首选项是适用于整个插件和功能块的全局设置，而属性是适用于单个资源的本地设置。本章探讨了Eclipse属性API并讨论了由开发者使用的Properties视图或Properties对话框访问属性的不同选择。本章还阐述了如何在工作区会话间保存属性。

参考文献

本书资源(2.9节).

Daum, Berthold, "Mutatis mutandis—Using Preference Pages as PropertyPages," October 24, 2003 (www.eclipse.org/articles/Article-Mutatis-mutandis/overlay-pages.html).

Johan, Dicky, "Take Control of Your Properties," May 20, 2003 (www.eclipse.org/articles/Article-Properties-View/properties-view.html).



第14章 构建器、标记和性质

增量式项目构造程序，也称为构建器 (builder)，当任意相关项目中的资源更改时将自动执行。比如，当创建或修改一个Java源文件时，Eclipse的增量Java编译器将为源文件添加注释并生成class文件。因为Java类文件可以由编辑器根据Java源文件完全重新生成，所以它们被称为派生资源 (derived resource)。

标记 (marker) 用于在资源内注解位置。比如，Eclipse Java编译器通过添加标记以表示编译错误、不建议的成员使用、书签等注释源文件。这些标记出现于左侧空白区域。当编辑Java文件时，和在Problems视图或Tasks视图的恰当时候，这些空白区域有时也称为装订线 (gutter)。

项目性质 (nature) 用于关联项目和构建器 (图14-1)。项目的Java性质使其成为一个Java项目并关联Eclipse增量式Java编译器。



图14-1 构建器和性质

本章的目标是讨论构建器、标记和性质。讨论将在Favorites产品的一个新的plugin.properties文件审计器中进行。属性审计器作为一个构建器和plugin.xml中的跨引用属性键实现。这些跨引用属性键包含plugin.properties文件中的条目。标记用于报告由审计器发现的问题。plugin.xml中的没有在plugin.properties文件中声明的键被标记为丢失 (missing)，而plugin.properties文件中未在plugin.xml文件中引用的键被标记为未使用过 (unused)。还创建一个新项目的性质以关联审计器和项目。

14.1 构建器

构建器的作用范围是项目内部。当项目中的一个或多个资源更改时，将通知与项目关联的构建器。如果批处理这些更改（参见9.3节），构建器收到一个包含所有已更改资源列表的通知，而不是每一个已更改资源都有一个独立通知。

提示 如果你需要一个不与任意特定项目关联的全局构建器，关联初期启动扩展点 (early startup extension point)（参见3.4.2节），并添加一个工作区更改监听器（参见9.1节）。该方法的不足之处是：无论是否真的需要，构建器都将消耗内存和操作周期。

构建器处理更改列表并通过一些步骤以更新它们的构建状态 (build state)（参见14.1节）。这些步骤包括：重新生成必需的派生资源，通知源资源等。在资源更改时通知构建器，比如当用户保存一个被修改过的Java源文件时，因此它将被频繁执行。由于该原因，构建器必须增量执行，表示它必须只重新构建那些已经更改的派生资源。

如果Eclipse Java编译器在每一次保存一个Java源文件时都重新构建项目中所有的源文件，那将使Eclipse性能显著降低。

14.1.1 声明构建器

创建plugin.properties审计器的第一个步骤包括添加一个构建器声明至Favorites插件清单。在Favorites的plugin.xml文件上打开插件清单编辑器，切换至Extensions页，添加一个org.eclipse.core.resources.builders扩展项（图14-2）。

点击org.eclipse.core.resources.builders扩展项以编辑它的属性，并为该扩展项设置id属性（图14-3）。

- id——“propertiesFileAuditor”

构建器的唯一标识符的最后一部分。如果声明出现于com.qualityeclipse.favorites插件中，那么，构建器的完全合格标识符为com.qualityeclipse.favorites.propertiesFileAuditor。

你可以右键点击扩展项并从上下文菜单中选择New > builder。构建器元素具有以下三个属性（图14-4）。

- hasNature——“true”。

表示构建器是否是由项目性质所占有的布尔值。如果值为true并且没有找到对应的性质，该构建器将不会运行，但将保留在项目的构建声明中。如果属性没有被指定，它将被假设为false。

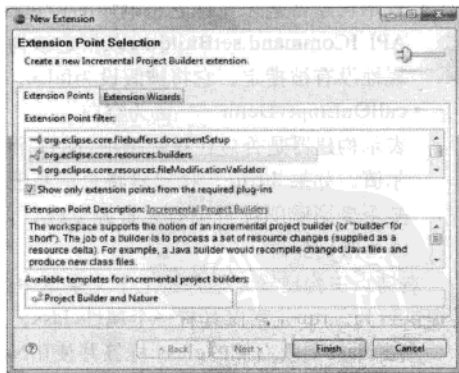


图14-2 选中org.eclipse.core.resources.builders扩展点的新建扩展项向导

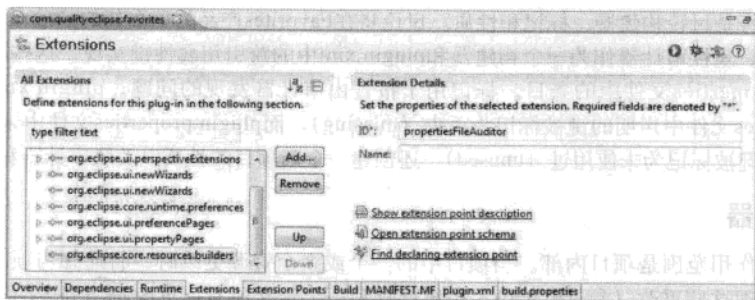


图14-3 显示构建器扩展项的插件清单编辑器

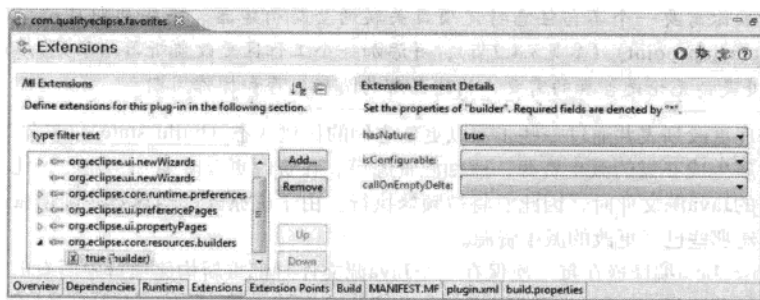


图14-4 显示构建器属性的插件清单编辑器

- **isConfigurable**——留为空白

表示构建器是否允许它将响应的构建触发器的自定义布尔值。如果为true，客户端将可以使用API `ICommand.setBuilding`以指定该构建器是否应根据一个特定的构建触发器而运行。如果该属性没有被指定，它将被假设为false。

- **callOnEmptyDelta**——留为空白

表示构建器是否应在被影响项目的资源增量为空时，基于INCREMENTAL_BUILD被调用的布尔值。如果为true，构建器将总是基于INCREMENTAL_BUILD类型的构建而被调用，而不管是否受影响的项目中是否有资源更改。如果为false或未指定，构建器将仅在受影响项目更改时才会被调用。

右键点击构建器元素并从上下文菜单中选择New > run以关联Java类和构建器。Java类将为构建器提供行为。run元素仅具有一个属性class，用于指定要执行的Java类。

点击class字段右侧的class:标签并使用New Java Class向导以在Favorites项目中创建一个具有指定包和类名称的新类。

- **class**——“com.qualityeclipse.favorites.builder.PropertiesFileAuditor”

org.eclipse.core.resources.IncrementalProjectBuilder的一个子类的完全合格名称。该类使用它的无参数构造函数进行初始化，但可以使用IExecutableExtension接口赋予参数（参见21.5.1节）。

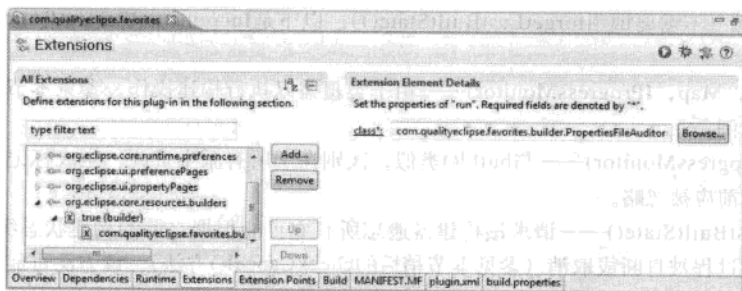


图14-5 显示运行属性的插件清单编辑器

Favorites插件清单的完整声明应与以下类似：

```
<extension
  id="propertiesFileAuditor"
  point="org.eclipse.core.resources.builders">
  <builder hasNature="true">
    <run class=
      "com.qualityeclipse.favorites.builder.PropertiesFileAuditor"/>
  </builder>
</extension>
```

14.1.2 IncrementalProjectBuilder

在14.1.1节的声明中指定的类必须是IncrementalProjectBuilder的子类，并最少应实现build()和clean()方法。build()方法将在以下时机由Eclipse调用：构建器应增量或完整构建相关文件和标记时。该方法具有几个参数以提供构建信息和用于向用户显示进度的机制。

- kind——当前被请求的构建类型。合法的值包括：FULL_BUILD、INCREMENTAL_BUILD和AUTO_BUILD。
- args——一个由参数名称作为键名的构建器特定的参数映射（键类型：String；值类型：String）或null，表示空映射。
- monitor——进度监视器，如果不需要进度报告和取消操作，则为null。

kind参数可具有以下值之一。

- FULL_BUILD——构建器应重新构建所有派生资源并如同它之前没有被执行过那样执行它的任务。
- CLEAN_BUILD——在执行一个完整构建之前，构建器应删除所有派生资源和标记（参见下面关于clean()方法的讨论）。
- INCREMENTAL_BUILD——构建器应仅重建那些需要被更新的派生资源，并仅执行根据它的优先构建状态所必需的任务。
- AUTO_BUILD——与INCREMENTAL_BUILD类似，区别在于构建是一个由增量构建自动触发之外（auto-building打开）。

当构建类型是CLEAN_BUILD时，调用IWorkspace.build()或IProject.build()将在调用构建类型为FULL_BUILD的build()方法之前触发clean()方法。clean()方法应忽略包括所有派生资源和所有类型为IMarker.PROBLEM的标记在内的已被计算为之前构建结果的附加状态。平台将处理忽略构建器的

上一次构建状态（不需要调用`forgetLastBuiltState()`）。以下是`IncrementalProjectBuilder`中几个有趣的方法。

- `build(int, Map, IProgressMonitor)`——由子类覆盖以执行构建操作。参见本节稍早的描述和稍后的示例。
- `clean(IProgressMonitor)`——与`build()`类似，区别在于所有派生资源、生成标记和之前的状态在构建之前应被忽略。
- `forgetLastBuiltState()`——请求该构建器遗忘所有它可能根据之前的构建状态缓存了的状态。如果构建过程被打断或取消（参见本节稍后的`checkCancel()`方法），该方法可能需要由一个子类调用。
- `getCommand()`——返回与该构建器关联的构建命令。该构建器可能包含项目特定的配置信息（参见14.1.4节）。
- `getDelta(IProject)`——返回上一次运行构建器之后，给定项目中记录更改的资源增量。如果没有可用的此类型的增量，则返回`null`。参见9.2节以了解关于处理资源更改事件的细节，也可以参考本节稍后的`shouldAudit()`方法。
- `getProject()`——返回与该构建器关联的项目。
- `isInterrupted()`——返回是否已经产生一个对于该构建的打断请求。后台自动构建将在另一线程尝试修改与构建线程并行的工作区时被打断。参见本节稍后的`shouldAudit()`方法。
- `setInitializationData(IConfigurationElement, String, Object)`——在使用构建器声明中指定的配置信息初始化构建器之后立即调用该方法（参见21.5节）。

在根据14.1.1节声明构建器之后，你必须实现`org.eclipse.core.resources.IncrementalProjectBuilder`的子类`PropertiesFileAuditor`，以执行该操作。当调用`build()`方法时，`PropertiesFileAuditor`构建器代表`shouldAudit()`以查看是否应执行一个审计，并在需要时，代表`auditPluginManifest()`以执行审计。

```
package com.qualityeclipse.favorites.builder;

import ...

public class PropertiesFileAuditor
    extends IncrementalProjectBuilder
{
    protected IProject[] build(
        int kind,
        Map args,
        IProgressMonitor monitor
    ) throws CoreException
    {
        if (shouldAudit(kind)) {
            auditPluginManifest(monitor);
        }
        return null;
    }
    ... other methods discussed later inserted here ...
}
```

`shouldAudit()`方法检查`FULL_BUILD`，或项目的`plugin.xml`或`plugin.properties`文件已经更改（参见9.2节）。如果一个构建器之前从未触发，那么`getDelta()`返回`null`。

```

private boolean shouldAudit(int kind) {
    if (kind == FULL_BUILD)
        return true;
    IResourceDelta delta = getDelta(getProject());
    if (delta == null)
        return false;
    IResourceDelta[] children = delta.getAffectedChildren();
    for (int i = 0; i < children.length; i++) {
        IResourceDelta child = children[i];
        String fileName = child.getProjectRelativePath().lastSegment();
        if (fileName.equals("plugin.xml")
            || fileName.equals("plugin.properties"))
            return true;
    }
    return false;
}
}

```

如果shouldAudit()方法确定清单和属性文件应当审计，那么将调用auditPluginManifest()方法以扫描plugin.xml和plugin.properties文件，并关联键/值对。plugin.xml中出现的所有键应在plugin.properties中具有一个对应的键/值对。在每一个长时间操作之前，检查是否构建已经被打断或取消。在每一个长时间操作之后，你应向用户报告进度（参见9.4节）。即使这不是严格必需的，但它无疑是礼貌的。如果你过早地退出你的构建进度，你可以需要在退出之前调用forgetLastBuildState()，以使在下次将执行一个完整重建。

```

public static final int MISSING_KEY_VIOLATION = 1;
public static final int UNUSED_KEY_VIOLATION = 2;

private void auditPluginManifest(IProgressMonitor monitor) {
    monitor.beginTask("Audit plugin manifest", 4);
    IProject proj = getProject();

    if (checkCancel(monitor))
        return;

    Map<String, Location> pluginKeys = scanPlugin(
        getProject().getFile("plugin.xml"));
    monitor.worked(1);

    if (checkCancel(monitor))
        return;
    Map<String, Location> propertyKeys = scanProperties(
        getProject().getFile("plugin.properties"));
    monitor.worked(1);

    if (checkCancel(monitor))
        return;

    Iterator<Map.Entry<String, Location>> iter
        = pluginKeys.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry<String, Location> entry = iter.next();
        if (!propertyKeys.containsKey(entry.getKey()))
            reportProblem(
                "Missing property key",
                ((Location) entry.getValue()),

```

```

        MISSING_KEY_VIOLATION,
        true);
    }

    monitor.worked(1);

    if (checkCancel(monitor))
        return;

    iter = propertyKeys.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry<String, Location> entry = iter.next();
        if (!pluginKeys.containsKey(entry.getKey()))
            reportProblem(
                "Unused property key",
                ((Location) entry.getValue()),
                UNUSED_KEY_VIOLATION,
                false);
    }
    monitor.done();
}

private boolean checkCancel(IProgressMonitor monitor) {
    if (monitor.isCanceled()) {
        // Discard build state if necessary.
        throw new OperationCanceledException();
    }

    if (isInterrupted()) {
        // Discard build state if necessary.
        return true;
    }
    return false;
}
}

```

auditPluginManifest()方法代表扫描plugin.xml和plugin.properties为两个独立的扫描方法。

```

private Map<String, Location> scanPlugin(IFile file) {
    Map<String, Location> keys = new HashMap<String, Location>();
    String content = readFile(file);
    int start = 0;
    while (true) {
        start = content.indexOf("\"", start);
        if (start < 0)
            break;
        int end = content.indexOf("'", start + 2);
        if (end < 0)
            break;
        Location loc = new Location();
        loc.file = file;
        loc.key = content.substring(start + 2, end);
        loc.charStart = start + 1;
        loc.charEnd = end;
        keys.put(loc.key, loc);
        start = end + 1;
    }
    return keys;
}

```



```
}
private Map<String, Location> scanProperties(IFile file) {
    Map<String, Location> keys = new HashMap<String, Location>();
    String content = readFile(file);
    int end = 0;
    while (true) {
        end = content.indexOf('=', end);
        if (end < 0)
            break;
        int start = end - 1;
        while (start >= 0) {
            char ch = content.charAt(start);
            if (ch == '\r' || ch == '\n')
                break;
            start--;
        }
        start++;
        String found = content.substring(start, end).trim();
        if (found.length() == 0
            || found.charAt(0) == '#'
            || found.indexOf('=') != -1)
            continue;
        Location loc = new Location();
        loc.file = file;
        loc.key = found;
        loc.charStart = start;
        loc.charEnd = end;
        keys.put(loc.key, loc);
        end++;
    }
    return keys;
}
```

下列两个扫描方法使用readFile()方法读取文件内容至内存。

```
private String readFile(IFile file) {
    if (!file.exists())
        return "";
    InputStream stream = null;
    try {
        stream = file.getContents();
        Reader reader =
            new BufferedReader(
                new InputStreamReader(stream));
        StringBuffer result = new StringBuffer(2048);
        char[] buf = new char[2048];
        while (true) {
            int count = reader.read(buf);
            if (count < 0)
                break;
            result.append(buf, 0, count);
        }
        return result.toString();
    }
    catch (Exception e) {
        FavoritesLog.logError(e);
        return "";
    }
}
```




```

        finally {
            try {
                if (stream != null)
                    stream.close();
            }
            catch (IOException e) {
                FavoritesLog.logError(e);
                return "";
            }
        }
    }
}

```

reportProblem()方法将一条消息附加至标准输出的末尾。在随后的几节中，该方法将被改进以生成标记作为替代（参见14.2.2节）。

```

private void reportProblem(
    String msg, Location loc, int violation, boolean isError
) {
    System.out.println(
        (isError ? "ERROR: " : "WARNING: ")
        + msg + " \"\"
        + loc.key + "\" in \"
        + loc.file.getFullPath());
    }
}

```

Location内部类被定义为一个不具有关联行为的内部数据持有者。

```

private class Location
{
    IFile file;
    String key;
    int charStart;
    int charEnd;
}

```

当关联至项目（参见14.1.4节和14.3.7节）时，构建器将把与以下内容类似的问题附加至标准输出的末尾。

```

ERROR: Missing property key "favorites.category.name"
      in /Test/plugin.xml
ERROR: Missing property key "favorites.view.name"
      in /Test/plugin.xml
WARNING: Unused property key "two"
      in /Test/plugin.properties
WARNING: Unused property key "three"
      in /Test/plugin.properties

```

14.1.3 派生资源

派生资源是可以由构建器完整生成的资源。Java类文件是派生资源，这是因为Java编译器可以从关联的Java源文件完整生成它们。当构建器创建派生资源时，它应使用IResource.setDerived()方法标记该文件为派生的。一个组提供者可以在稍后假设该文件默认不需要接受版本控制。

- setDerived(boolean)——设置该资源子类是否被标记为派生的。该操作不产生一个资源更改事件，并且不触发自动构建。

14.1.4 关联构建器与项目

使用性质以关联构建器和项目是首选方法（参见14.3节），但你可以不使用性质关联构建器与项

目。你可以在工作台窗口中创建一个命令（参见6.6.6节）以调用以下的addBuilderToProject()方法以关联你的审计器和当前选中项目。或者，你可以在启动时，在工作台的所有项目间循环，并调用以下addBuilderToProject()方法。如果你不使用项目性质，那么请确认设置hasNature属性为false（图14-4）。

使用命令处理器还是使用项目性质来关联构建器与项目没有什么区别。但在这里，你将创建一个项目性质以实现关联（参见14.3节）。将以下代码放置于收藏夹PropertiesFileAuditor类中。

```
public static final String BUILDER_ID =
    FavoritesActivator.PLUGIN_ID + ".propertiesFileAuditor";

public static void addBuilderToProject(IProject project) {

    // Cannot modify closed projects.
    if (!project.isOpen())
        return;
    // Get the description.
    IProjectDescription description;
    try {
        description = project.getDescription();
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }

    // Look for builder already associated.
    ICommand[] cmds = description.getBuildSpec();
    for (int j = 0; j < cmds.length; j++)
        if (cmds[j].getBuilderName().equals(BUILDER_ID))
            return;
    // Associate builder with project.
    ICommand newCmd = description.newCommand();
    newCmd.setBuilderName(BUILDER_ID);
    List<ICommand> newCmds = new ArrayList<ICommand>();
    newCmds.addAll(Arrays.asList(cmds));
    newCmds.add(newCmd);
    description.setBuildSpec(
        (ICommand[]) newCmds.toArray(
            new ICommand[newCmds.size()]));
    try {
        project.setDescription(description, null);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
}
```

每一个工作台项目均包含一个.project文件（参见1.4.2节）。该文件包含构建命令。执行该方法将使以下内容出现于项目的.project文件的buildSpec部分。

```
<buildCommand>
  <name>
    com.qualityeclipse.favorites.propertiesFileAuditor
  </name>
  <arguments>
  </arguments>
</buildCommand>
```

除了addBuilderToProject()方法之外, 你将需要一个对应的removeBuilderFromProject()方法:

```
public static void removeBuilderFromProject(IProject project) {

    // Cannot modify closed projects.
    if (!project.isOpen())
        return;

    // Get the description.
    IProjectDescription description;
    try {
        description = project.getDescription();
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }

    // Look for builder.
    int index = -1;
    ICommand[] cmds = description.getBuildSpec();
    for (int j = 0; j < cmds.length; j++) {
        if (cmds[j].getBuilderName().equals(BUILDER_ID)) {
            index = j;
            break;
        }
    }
    if (index == -1)
        return;

    // Remove builder from project.
    List<ICommand> newCmds = new ArrayList<ICommand>();
    newCmds.addAll(Arrays.asList(cmds));
    newCmds.remove(index);
    description.setBuildSpec(
        (ICommand[]) newCmds.toArray(
            new ICommand[newCmds.size()]));
    try {
        project.setDescription(description, null);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
}
```

14.1.5 触发构建器

一般地, 项目的构建过程由用户选择一个构建操作, 或由工作台中作为资源更改响应的自动构建而触发。如果需要, 你可以使用以下方法之一在程序中触发构建过程:

IProject

- build(int, IProgressMonitor)——在项目上运行构建过程, 使得所有关联的构建器运行。第一个参数表示构建类型, FULL_BUILD、INCREMENTAL_BUILD或CLEAN_BUILD (参见14.1.2节)。
- build(int, String, Map, IProgressMonitor)——触发一个构建器运行项目。第一个参数表示构

建类型, FULL_BUILD、INCREMENTAL_BUILD或CLEAN_BUILD (参见14.1.2节), 而第二个指定将运行的是哪一个构建器。

IWorkspace

build(int, IProgressMonitor)——在工作区的所有打开项目上运行构建过程。第一个参数表示构建类型, FULL_BUILD、INCREMENTAL_BUILD或CLEAN_BUILD (参见14.1.2节)。

14.2 标记

标记用于在资源内标识指定位置。比如, Eclipse Java编译器不仅仅从源文件生成类文件, 它通过添加标记标识源文件以表示编译错误、不建议的代码使用等。标记不修改它们所标识的资源, 但作为替代, 它们存储于工作区元数据区域。标记由编辑器自动更新, 以当用户编辑文件时, 它们可以被恰当的重新定位或删除。不向控制台发送消息, 你需要PropertiesFileAuditor以创建一个标记, 表示问题存在的位置 (图14-6)。



图14-6 标记声明与数据结构

14.2.1 标记类型

标记根据它的类型进行分组。每一个标记类型具有一个标识符和零个或以上的父标记类型, 而不是行为。新的标记类型根据已有的进行声明。由org.eclipse.core.resources插件添加的标记类型作

为常量出现于IMarker中，并包括：

- org.eclipse.core.resources.bookmark—IMarker.BOOKMARK——出现于Bookmarks视图中的标记的超类型。
- org.eclipse.core.resources.marker—IMarker.MARKER——所有标记的根超类型。
- org.eclipse.core.resources.problemmarker—IMarker.PROBLEM——出现于Problems视图中的标记的超类型。
- org.eclipse.core.resources.taskmarker—IMarker.TASK——出现于Tasks视图中的标记的超类型。
- org.eclipse.core.resources.textmarker—IMarker.TEXT——所有基于文本的标记的超类型。

对于当前而言，你需要为插件清单审计结果引入一个新的标记类型。切换至插件清单编辑器的Extensions页，点击Add...按钮以添加一个org.eclipse.core.resources.markers扩展项。在左侧的树中选择刚添加的扩展项并在右侧的字段中指定“auditmarker”作为ID，“Properties Auditor Marker”作为名称（图14-7）。

你需要你的标记出现于Problems视图中，所以，通过右键点击标记的扩展项并选择New > super以指定org.eclipse.core.resources.problemmarker作为超类型。点击新建的super元素并输入“org.eclipse.core.resources.problemmarker”作为类型属性。标记与插件清单中的一些源代码或插件属性文件有关，所以使用同样的步骤指定org.eclipse.core.resources.textmarker。

为了使标记在多个会话间保留，右键点击markers声明，并选择New > persistent。点击新建的persistent元素并输入“true”作为值属性。

你从稍早指定的标记超类型继承几个标记属性，但你还需要关联两个新的属性至审计标记。右键点击markers声明并选择New > persistent。点击新建的attribute属性并输入“key”作为值属性。重复该步骤以指定“violation”属性。

当完成了这些任务后，新的标记类型声明与以下类型类似：

```
<extension
  id="auditmarker"
  point="org.eclipse.core.resources.markers"
  name="Properties Auditor Marker">
  <super type="org.eclipse.core.resources.problemmarker"/>
  <super type="org.eclipse.core.resources.textmarker"/>
  <attribute name="key"/>
  <attribute name="violation"/>
  <persistent value="true"/>
</extension>
```

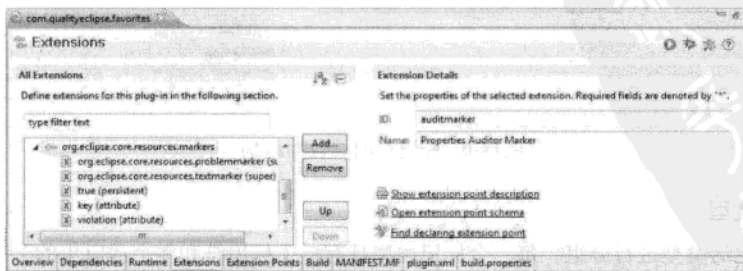


图14-7 显示选中标记扩展点的新建扩展项向导

上面的声明指定了标记的本地标识符。完整标识符是插件标识符加上作为一个常量添加于 PropertiesFileAuditor 的本地标识符。

```
private static final String MARKER_ID =  
    FavoritesActivator.PLUGIN_ID + ".auditmarker";
```

14.2.2 创建并删除标记

你需要为每一个发现的问题创建一个标记，但首先你必须删除所有老的标记。为了完成该任务，添加以下代码至 auditPluginManifest() 方法：

```
private void auditPluginManifest(IProgressMonitor monitor) {  
    monitor.beginTask("Audit plugin manifest", 4);  
  
    if (!deleteAuditMarkers(getProject())) {  
        return;  
    }  
  
    if (checkCancel(monitor)) {  
        return;  
    }  
    ... etc ...  
}
```

该方法调用以下新方法以删除指定项目中所有已有标记。

```
public static boolean deleteAuditMarkers(IProject project) {  
    try {  
        project.deleteMarkers(  
            MARKER_ID, false, IResource.DEPTH_INFINITE);  
        return true;  
    }  
    catch (CoreException e) {  
        FavoritesLog.logError(e);  
        return false;  
    }  
}
```

接下来，添加两个常量，并修改 reportProblem() 方法（参见 14.1.2 节）以创建标记和设置标记属性（参见 14.2.3 节）来表示问题。经过改进的方法不仅创建标记，还设置不同的标记属性。这些属性将在下一章中讨论。

```
public static final String KEY = "key";  
public static final String VIOLATION = "violation";  
  
private void reportProblem(  
    String msg, Location loc, int violation, boolean isError)  
{  
    try {  
        IMarker marker = loc.file.createMarker(MARKER_ID);  
        marker.setAttribute(IMarker.MESSAGE, msg + ": " + loc.key);  
        marker.setAttribute(IMarker.CHAR_START, loc.charStart);  
        marker.setAttribute(IMarker.CHAR_END, loc.charEnd);  
        marker.setAttribute(  
            IMarker.SEVERITY,  
            isError  
                ? IMarker.SEVERITY_ERROR  
                : IMarker.SEVERITY_WARNING);
```

```

        marker.setAttribute(KEY, loc.key);
        marker.setAttribute(VIOLATION, violation);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }
}

```

最后，创建并设置属性，而删除标记将生成资源更改事件。基于效率考虑，修改build()方法以封装在IWorkspaceRunnable中对auditPluginManifest()的调用，以使事件将得到批处理并在操作完成时传递（参见9.3节）：

```

protected IProject[] build(
    int kind,
    Map args,
    IProgressMonitor monitor
) throws CoreException
{
    if (shouldAudit(kind)) {
        ResourcesPlugin.getWorkspace().run(
            new IWorkspaceRunnable() {
                public void run(IProgressMonitor monitor)
                    throws CoreException
                {
                    auditPluginManifest(monitor);
                }
            },
            monitor
        );
    }
    return null;
}

```

当完成了以上工作后，由PropertiesFileAuditor报告的问题将出现Problems视图中而不是Console视图（图14-8）。此外，标记作为小的警告和错误图标出现于plugin.xml和plugin.properties编辑器的左侧。

14.2.3 标记属性

标记属性以键/值对的形式出现。键是一个字符串，而值可以是字符串、整数或布尔值。用于访问属性的IMarker方法包括：

- getAttribute(String)——返回具有给定名称的属性。返回结果是一个string、integer或Boolean的实例。如果未定义该属性则返回null。
- getAttribute(String, boolean)——返回具有给定名称的布尔值的属性。如果定义了属性，标记不存在时，或它不是一个布尔值时，返回给定默认值。
- getAttribute(String, int)——返回具有给定名称的整数值的属性。如果属性未定义，标记不存在，或它不是一个整数时，返回给定默认值。
- getAttribute(String, String)——返回具有给定名称的字符串值的属性。如果属性未定义，标记不存在，或它不是一个字符串值时，返回给定默认值。

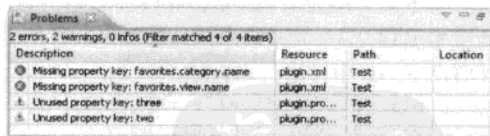


图14-8 问题视图包含由审计器发现的问题

- `getAttributes()`——返回标记的属性映射。映射具有字符串键和`string`、`integer`、`Boolean`或`null`类型的值。如果标记没有属性，则返回`null`。
- `getAttributes(String[])`——返回具有给定名称的属性。返回结果是一个数组。它的元素对应于具有给定属性名称数组的元素。每一个元素是`string`、`integer`、`Boolean`或`null`类型。
- `setAttribute(String, boolean)`——设置具有给定名称的布尔值的属性。该方法更改资源。这些更改将被报告于一个后续资源更改事件，包括一个说明该标记已经被修改的表示。
- `setAttribute(String, int)`——设置具有给定名称的整数值属性的属性。该方法更改资源。这些更改将被报告于一个后续资源更改事件，包括一个说明该标记已经被修改的表示。
- `setAttribute(String, Object)`——设置具有给定名称的属性。该值必须是`string`、`integer`、`Boolean`或`null`类型。如果值为`null`，该属性将被认为是未定义的。该方法更改资源。这些更改将被报告于一个后续资源更改事件，包括一个说明该标记已经被修改的表示。
- `setAttributes(String[], Object[])`——设置该标记中的给定属性键/值对。值必须是`string`、`integer`、`Boolean`或`null`类型。如果值为`null`，属性的新值被认为是未定义的。该方法更改资源。这些更改将被报告于一个后续资源更改事件，包括一个说明该标记已经被修改的表示。
- `setAttributes(Map)`——设置该标记的属性为给定映射包含的属性。属性必须是`string`、`integer`或`Boolean`的实例。标记中之前设置的而未包含在给定映射中的属性被认为将被移除。传递一个`null`参数等于移除所有标记属性。该方法更改资源。这些更改将被报告于一个后续资源更改事件，包括一个说明该标记已经被修改的表示。

基于文档目的，标记属性在插件清单中声明，但不在编译或执行时使用。比如，在标记类型声明中，为标记类型声明了两个新属性，名为`key`和`violation`。作为选择，他们可以使用XML `<!-- -->` 语句记录，但我们推荐使用下面的`attribute`声明，由于Eclipse的后续版本可能使用到他们。

```
<extension
  id="auditmarker"
  point="org.eclipse.core.resources.markers"
  name="Properties Auditor Marker">
  <super type="org.eclipse.core.resources.problemmarker"/>
  <super type="org.eclipse.core.resources.textmarker"/>
  <persistent value="true"/>
  <attribute name="key"/>
  <attribute name="violation"/>
</extension>
```

`org.eclipse.core.resources`插件引入了几个在Eclipse中普遍使用的属性。下列属性键在IMarker中定义。

- `CHAR_END`——字符结尾标记属性。整数值表示文本标记结束的位置。该属性对于文件以0为基数，并且是唯一的。
- `CHAR_START`——字符开始标记属性。整数值表示文本标记开始的位置。该属性对于文件以0为基数，并且是唯一的。
- `DONE`——完成标记属性。布尔值表示标记（如任务）是否被认为已完成。
- `LINE_NUMBER`——行数标记属性。整数值表示文本标记的行数。该属性以1为基数。
- `LOCATION`——位置标记属性。位置是一个可读（本地化）字符串。该字符串可以用于区分资源的不同标记。它本身应是简明的，并以用户为目标的。该属性的内容和形式不由平台指定或解释。

- **MESSAGE**——消息标记属性。一个本地化的字符串描述标记的性质（比如，一个书签或任务的名称）。该属性的内容和形式不由平台指定或解释。
- **PRIORITY**——优先级标记属性。一个来源于IMarker中定义的常量集的数字：PRIORITY_HIGH、PRIORITY_LOW和PRIORITY_NORMAL。
- **SEVERITY**——严重性标记属性。一个来源于IMarker中定义的常量集的数字：SEVERITY_ERROR、SEVERITY_WARNING和SEVERITY_INFO。
- **TRANSIENT**——暂时程序标记属性。一个表示即使它的类型被声明为持久时，标记（比如，任务）是否被认为是暂时的布尔值。
- **USER_EDITABLE**——用户可编辑的标记属性。一个表示是否用户可以手动更改标记（比如，任务）的布尔值。默认为true。

在改进的reportProblem()方法中（参见14.2.2节），设置了几个标记属性。这些标记属性稍后由Eclipse解释。Problems视图使用IMarker.MESSAGE和IMarker.MESSAGE属性以操作描述（Description）和位置（Location）列。编辑器使用IMarker.CHAR_START和IMarker.CHAR_END属性以决定多大范围的文本应被高亮显示。

14.2.4 标记解析——快速修复

当你可以生成标记时，用户可以通过双击在问题中的对应条目快速跳转至问题的位置，但这对于修复所提供的问题没有益处。使用标记解析，你可以提供一个用于修复你的构建器标明的问题的自动机制。

创建一个新的org.eclipse.ui.ide.marker - Resolution扩展项（图14-9），添加一个markerResolution Generator嵌套元素（图14-10），并指定标记类型为“com.qualityeclipse.favorites.auditmarker”。

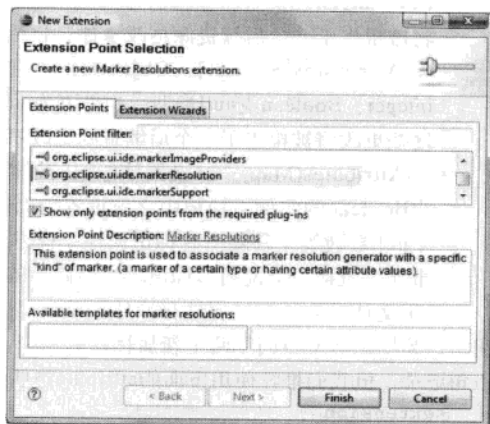


图14-9 显示选中org.eclipse.ui.ide.marker Resolution扩展点的新建扩展项向导

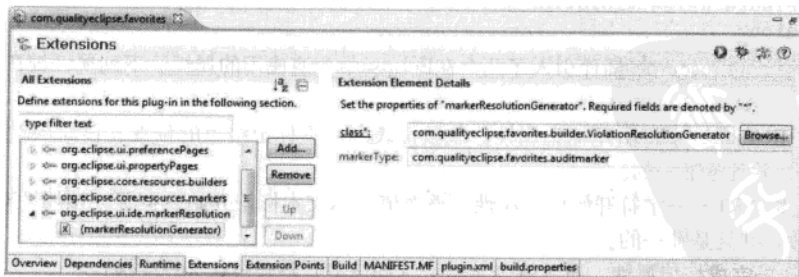


图14-10 显示markerResolutionGenerator属性的插件清单编辑器

使用New Java Class向导以在com.qualityeclipse.favorites.builder包中生成一个名为ViolationResolutionGenerator的标记解析类。当完成时，声明应与以下内容类似：

```
<extension point="org.eclipse.ui.ide.markerResolution">
  <markerResolutionGenerator
    markerType="com.qualityeclipse.favorites.auditmarker"
    class="com.qualityeclipse.favorites.builder
      .ViolationResolutionGenerator">
  </markerResolutionGenerator>
</extension>
```

ViolationResolutionGenerator类为用户提供了对于任意com.qualityeclipse.favorites.auditmarker标记的可能解决办法。该功能通过使用org.eclipse.ui.IMarkerResolutionGenerator2接口实现(IMarkerResolutionGenerator2接口从Eclipse 3.0开始引入。它提供了附加功能并取代当前不建议使用的IMarkerResolutionGenerator)。

```
package com.qualityeclipse.favorites.builder;

import ...

public class ViolationResolutionGenerator
  implements IMarkerResolutionGenerator2
{
  public boolean hasResolutions(IMarker marker) {
    switch (getViolation(marker)) {
      case PropertiesFileAuditor.MISSING_KEY_VIOLATION :
        return true;
      case PropertiesFileAuditor.UNUSED_KEY_VIOLATION :
        return true;
      default :
        return false;
    }
  }

  public IMarkerResolution[] getResolutions(IMarker marker){
    List<IMarkerResolution2> resolutions
      = new ArrayList<IMarkerResolution2>();
    switch (getViolation(marker)) {
      case PropertiesFileAuditor.MISSING_KEY_VIOLATION :
        resolutions.add(
          new CreatePropertyKeyResolution());
        break;
      case PropertiesFileAuditor.UNUSED_KEY_VIOLATION :
        resolutions.add(
          new DeletePropertyKeyResolution());
        resolutions.add(
          new CommentPropertyKeyResolution());
        break;
      default :
        break;
    }

    return (IMarkerResolution[]) resolutions.toArray(
      new IMarkerResolution[resolutions.size()]);
  }

  private int getViolation(IMarker marker) {
    return marker.getAttribute(PropertiesFileAuditor.VIOLATION, 0);
  }
}
```

ViolationResolutionGenerator类返回一个或多个org.eclipse.ui.IMarkerResolution2的实例（与IMarkerResolutionGenerator2类似，IMarkerResolution2由Eclipse 3.0引入，以替代现在不建议使用的IMarkerResolution），以表示用于冲突的可能解决办法。比如，返回CreatePropertyKeyResolution的实例以用于丢失的属性键冲突：

```
package com.qualityeclipse.favorites.builder;

import ...

public class CreatePropertyKeyResolution
    implements IMarkerResolution2
{
    public String getDescription() {
        return "Append a new property key/value pair"
            + " to the plugin.properties file";
    }

    public Image getImage() {
        return null;
    }

    public String getLabel() {
        return "Create a new property key";
    }
}
```

如果用户选择该解决办法，将执行run()方法，打开或激活属性编辑器并在末尾添加一个新的属性键/值对。

```
public void run(IMarker marker) {

    // Get the corresponding plugin.properties.
    IFile file = marker.getResource().getParent().getFile(
        new Path("plugin.properties"));
    if (!file.exists()) {
        ByteArrayInputStream stream =
            new ByteArrayInputStream(new byte[] {});
        try {
            file.create(stream, false, null);
        }
        catch (CoreException e) {
            FavoritesLog.logError(e);
            return;
        }
    }
    // Open or activate the editor.
    IWorkbenchPage page = PlatformUI.getWorkbench()
        .getActiveWorkbenchWindow().getActivePage();

    IEditorPart part;
    try {
        part = IDE.openEditor(page, file, true);
    }
    catch (PartInitException e) {
        FavoritesLog.logError(e);
    }
}
```



```
        return;
    }

    // Get the editor's document.
    if (!(part instanceof ITextEditor)) {
        return;
    }

    ITextEditor editor = (ITextEditor) part;
    IDocument doc = editor.getDocumentProvider()
        .getDocument(new FileEditorInput(file));

    // Determine the text to be added.
    String key;
    try {
        key = (String) marker.getAttribute(PropertiesFileAuditor.KEY);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }

    String text = key + "=Value for " + key;

    // If necessary, add a newline.
    int index = doc.getLength();
    if (index > 0) {
        char ch;
        try {
            ch = doc.getChar(index - 1);
        }
        catch (BadLocationException e) {
            FavoritesLog.logError(e);
            return;
        }

        if (ch != '\r' || ch != '\n') {
            text = System.getProperty("line.separator") + text;
        }
    }

    // Append the new text.
    try {
        doc.replace(index, 0, text);
    }
    catch (BadLocationException e) {
        FavoritesLog.logError(e);
        return;
    }

    // Select the value so the user can type.
    index += text.indexOf('=') + 1;
    editor.selectAndReveal(index, doc.getLength() - index);
}
```

14.2.5 查找标记

你可以向资源查询所有它的标记或给定类型的所有标记。如果资源是一个容器，如文件夹、项

目或工作区根，你还可以查询该容器的后代的所有标记。深度可以是0（仅该容器）、1（容器和它的直接后代）或无穷大（资源和所有的直接和间接后代）。比如，为了获取与一个文件夹相关的所有标记和它所有的后代，你可以使用一个如下所示的表达式：

```
IMarker[] markers;
try {
    markers = myFolder.findMarkers(
        IMarker.PROBLEM, true, IResource.DEPTH_INFINITE);
}
catch (CoreException e) {
    // Log the exception and bail out.
}
```

14.3 性质

性质用于关联项目和功能，如构建器、工具或进程。性质还可以用于决定一个操作是否应为可见的（参见6.7.2节）。尽管标记只具有有限的功能，但它可以应用于任意资源。性质被设计用于包含附加功能，但它只能应用于项目。标记仅适用于单个工作区的一个资源，而性质是项目的一部分，因此它可以由多个开发者共享。

Java性质使得一个项目成为一个Java项目，将它与所有其他类型的项目区分开来。当性质，如Java性质，被添加一个项目时，项目.project的文件（参见1.4.2节）将被修改以包含性质的标识符（参见14.1.4节），并且性质有机会对项目进行配置。

比如，Java性质通过添加Java编译器作为一个构建命令设置项目。性质还使得项目被工作台区别对待。比如，仅具有Java性质的项目由Package Explorer显示。当移除一个性质时，它有机会取消设置或从项目移除它自身的相关内容。以下是一些在Eclipse内部定义的性质。它们提供了不同类型的行为。

- org.eclipse.jdt.core.javanature——关联Eclipse增量式Java编译器与项目，并使得项目出现于Java相关的视图中，如Package Explorer视图。
- org.eclipse.pde.PluginNature——将插件清单和扩展点模式构建器关联至项目，验证plugin.xml文件的内容并基于项目的插件依赖声明（参见2.3.1节）更新项目的Java构建路径。
- org.eclipse.pde.FeatureNature——关联功能部件构建器与项目，并验证feature.xml文件的内容（参见18.1.2节）。
- org.eclipse.pde.UpdateSiteNature——关联站点构建器与项目，验证site.xml文件的内容（参见18.3.2节）。

14.3.1 声明性质

对于Favorites产品而言，你需要一个新的propertiesAuditor性质以关联属性文件审计构建器与项目。我们从在Favorites插件清单创建一个新的org.eclipse.core.resources.natures扩展项开始。切换至Extensions页，点击Add...按钮，选择org.eclipse.core.resources.natures，然后点击Finish（图14-11）。

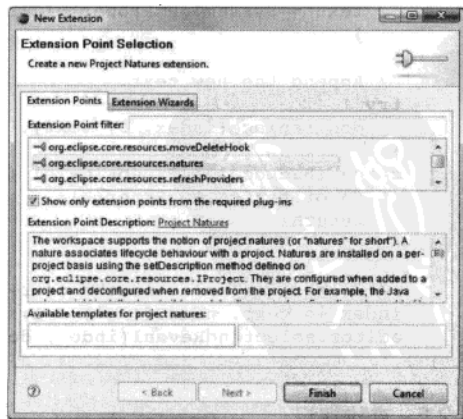


图14-11 显示选中性质扩展点的新建扩展项向导

点击新键的扩展项以编辑属性，更改id为“propertiesAuditor”，并更改name为“Favorites Properties Auditor”（图14-12）。性质声明应与以下内容类似：

```
<extension
  id="propertiesAuditor"
  name="Favorites Properties Auditor"
  point="org.eclipse.core.resources.natures">
</extension>
```

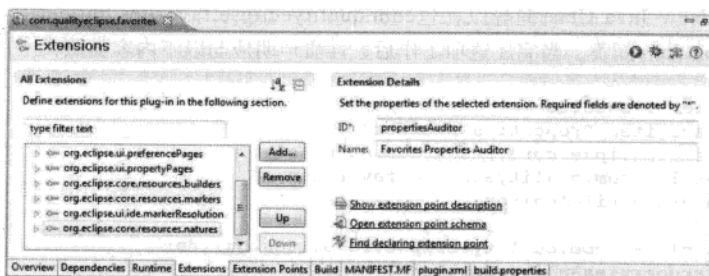


图14-12 显示性质的属性的扩展项细节

与构建声明类似，性质声明包含性质的本地标识符。性质的完整标识符是包含性质的插件标识符连接性质的本地标识符。在这里是com.qualityeclipse.favorites.propertiesAuditor。

14.3.2 关联构建器与性质

现在，你需要关联你的构建器与性质。点击org.eclipse.core.resources.natures扩展点并选择New > builder。输入构建器id。在这里是“com.qualityeclipse.favorites.propertiesFileAuditor”（图14-13）。

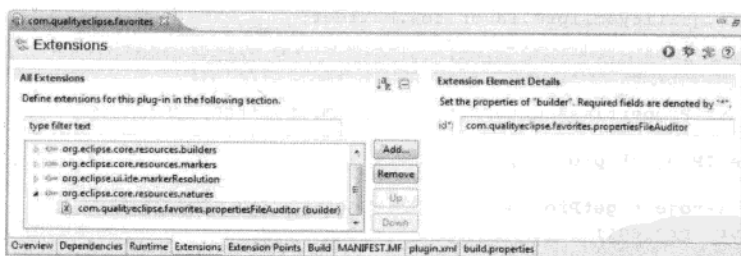


图14-13 显示构建器属性的扩展项元素细节

此外，返回至构建器声明（参见14.1.1节）并修改hasNature属性为“true”。在完成这项工作后，性质声明应与以下内容类似：

```
<extension
  id="propertiesAuditor"
  name="Favorites Properties Auditor"
  point="org.eclipse.core.resources.natures">
  <builder id="com.qualityeclipse.favorites
    .propertiesFileAuditor"/>
</extension>
```

这些更改确保如果性质没有与项目关联时，构建器将从项目的构建步骤中忽略。如果你想要你

的构建器无论你的性质是否显示都工作，那么将这些内容从你的性质的声明中去掉。

14.3.3 IProjectNature

性质可以具有行为以设置和取消设置项目。与Java性质类似，你需要性质以添加你的构建器至项目的构建说明。你可以右键点击org.eclipse.core.resources.natures扩展点并选择New > runtime，然后右键点击(runtime)嵌套元素并选择New > run。在插件清单编辑器中，点击class字段左侧的“class”标签，然后使用New Java Class向导以在包com.qualityeclipse.favorites.builder中生成一个名为的PropertiesAuditorNature新类。当完成这项工作后，性质声明应与以下内容类似：

```
<extension
    id="propertiesAuditor"
    name="Favorites Properties Auditor"
    point="org.eclipse.core.resources.natures">
    <builder id="com.qualityeclipse.favorites
        .propertiesFileAuditor"/>
    <runtime>
        <run class="com.qualityeclipse.favorites.builder
            .PropertiesAuditorNature"/>
    </runtime>
</extension>
```

在性质声明中指定的类必须实现org.eclipse.core.resources.IProjectNature接口。当添加性质至项目时，该类被初始化，并调用setProject()方法，然后是configure()方法。当性质从项目中移除时，调用deconfigure()方法。

与Java性质类似，你需要性质通过addBuilderToProject()方法（参见14.1.4节）以添加构建器至项目的构建说明，并在项目配置好后，触发后台构建（参见21.8节）。当性质从项目中移除时，构建说明被修改，并且所有审计标记都被移除。

```
package com.qualityeclipse.favorites.builder;

import ...

public class PropertiesAuditorNature implements IProjectNature
{
    private IProject project;

    public IProject getProject() {
        return project;
    }
    public void setProject(IProject project) {
        this.project = project;
    }

    public void configure() throws CoreException {
        PropertiesFileAuditor.addBuilderToProject(project);
        new Job("Properties File Audit") {
            protected IStatus run(IProgressMonitor monitor) {
                try {
                    project.build(
                        PropertiesFileAuditor.FULL_BUILD,
                        PropertiesFileAuditor.BUILDER_ID,
                        null,
                        monitor);
                }
            }
        }.runAndDispose(monitor);
    }
}
```

```

    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
    return Status.OK_STATUS;
}
}.schedule();
}

public void deconfigure() throws CoreException {
    PropertiesFileAuditor.removeBuilderFromProject(project);
    PropertiesFileAuditor.deleteAuditMarkers(project);
}
}
}

```

14.3.4 必需的性质

一个性质对另一个性质的依赖可以在性质的声明中表示（参见14.3.1节）。当必需的性质没有显示或不可用时，Eclipse使具有要求的性质不可用。比如，propertiesAuditor性质依赖于Java性质和PDE性质。如果你打算在你的性质的声明中表达它，它将与以下内容类似：

```

<extension
  id="propertiesAuditor"
  name="Favorites Properties Auditor"
  point="org.eclipse.core.resources.natures">
  <builder id="com.qualityeclipse.favorites
    .propertiesFileAuditor">
  </builder>
  <runtime>
    <run class="com.qualityeclipse.favorites.builder
      .PropertiesAuditorNature"/>
  </runtime>
  <requires-nature id="org.eclipse.jdt.core.javanature"/>
  <requires-nature id="org.eclipse.pde.PluginNature"/>
</extension>

```

14.3.5 冲突的性质

一个性质与另一个或多个性质的冲突也可以在性质的声明中表示。在你的性质声明中，添加一个one-of-nature嵌套元素。该元素指定了一个性质集的名称。如果任意其他性质在one-of-nature嵌套元素中指定同样的字符串并被添加至与你的性质相同的项目中，那么Eclipse将使得这两个性质都不可用。

```

<extension
  id="propertiesAuditor"
  name="Favorites Properties Auditor"
  point="org.eclipse.core.resources.natures">
  <builder id="com.qualityeclipse.favorites
    .propertiesFileAuditor">
  </builder>
  <runtime>
    <run class="com.qualityeclipse.favorites.builder
      .PropertiesAuditorNature"/>
  </runtime>
  <requires-nature id="org.eclipse.jdt.core.javanature"/>
  <requires-nature id="org.eclipse.pde.PluginNature"/>
  <one-of-nature id="pluginAuditors">
</extension>

```


14.3.6 性质图像

项目性质可以使用org.eclipse.ui.ide.projectNatureImages扩展点以拥有一个相关的图像。指定的图像显示于标准项目图像的右上角。比如, org.eclipse.jdt.ui插件关联一个“J”的图像与Java性质, 因此所有Java项目的图标在右上角具有一个蓝色的小“J”。

```
<extension point="org.eclipse.ui.ide.projectNatureImages">
  <image
    icon="icons/full/ovrl6/java_ovr.gif"
    natureId="org.eclipse.jdt.core.javanature"
    id="org.eclipse.ui.javaProjectNatureImage"/>
</extension>
```

这里的性质没有定义项目类型, 甚至于关联属性审计工具与项目, 因此提供一个项目性质图像是不合适的。

14.3.7 关联性质与项目

与管理构建器和项目类似(参见14.1.4节), 你可以通过修改项目的描述关联性质与项目。为了阐述这一点, 构建一个命令为项目切换propertiesAuditor性质。首先, 在最高级收藏夹菜单中声明一个新的动态菜单添加项(参见6.2.1节)。

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
    locationURI="menu:org.eclipse.ui.main.menu?after=additions">
    <menu
      id="com.qualityeclipse.favorites.menus.favoritesMenu"
      label="Favorites"
      mnemonic="v">
      ... existing menu commands declarations here ...
      <dynamic
        id="com.qualityeclipse.favorites.menus.toggleProjectNature">
          class="com.qualityeclipse.favorites.contributions
            .ToggleProjectNatureContributionItem"
        </dynamic>
      </menu>
```

然后, 创建MenuContribution(参见7.3.2节)。它检查与每一个被选中项目关联的性质, 并添加propertiesAuditor性质至每一个不具有与它关联的性质的项目, 并从所有其他选中项目中移除该性质。一般地, 性质被添加至或从项目中移除作为一个更大步骤(如创建Java项目)的一部分, 但该菜单添加项足够显示它是如何被完成的机制。

```
public class ToggleProjectNatureContributionItem
  extends ContributionItem
{
  public ToggleProjectNatureContributionItem() {
  }
  public ToggleProjectNatureContributionItem(String id) {
    super(id);
  }
  ... subsequent methods go here ...
}
```

fillMenu方法在Favorites菜单第一次生成以创建菜单项时被调用。该方法还添加监听器至新的

将在菜单项被选中时调用的菜单项。此外，我们还必须添加一个监听器至菜单本身，以使菜单项可以在每一次菜单将要显示时更新它的状态。

```
public void fill(Menu menu, int index) {
    final MenuItem menuItem = new MenuItem(menu, SWT.CHECK, index);
    menuItem.setText("Add/Remove propertiesAuditor project nature");
    menuItem.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            run();
        }
    });
    menu.addMenuListener(new MenuAdapter() {
        public void menuShown(MenuEvent e) {
            updateState(menuItem);
        }
    });
}
```

当Favorites菜单将要显示时，将调用menuShown方法，给updateState方法一个机会以更新我们的菜单项。

```
protected void updateState(MenuItem menuItem) {
    Collection<IProject> projects = getSelectedProjects();
    boolean enabled = projects.size() > 0;
    menuItem.setEnabled(enabled);
    menuItem.setSelection(enabled &&
        PropertiesAuditorNature.hasNature(projects.iterator().next()));
}

private Collection<IProject> getSelectedProjects() {
    Collection<IProject> projects = new HashSet<IProject>();
    IWorkbenchWindow window =
        PlatformUI.getWorkbench().getActiveWorkbenchWindow();
    ISelection selection = window.getActivePage().getSelection();
    if (selection instanceof IStructuredSelection) {
        for (Iterator<?> iter = ((IStructuredSelection)
            selection).iterator(); iter.hasNext();) {
            // translate the selected object into a project
            Object elem = iter.next();
            if (!(elem instanceof IResource)) {
                if (!(elem instanceof IAdaptable))
                    continue;
                elem = ((IAdaptable) elem).getAdapter(IResource.class);
                if (!(elem instanceof IResource))
                    continue;
            }
            if (!(elem instanceof IProject)) {
                elem = ((IResource) elem).getProject();
                if (!(elem instanceof IProject))
                    continue;
            }
            projects.add((IProject) elem);
        }
    }
    return projects;
}
```

当选中菜单项时，将调用widgetSelected方法。该方法调用run方法以从选中项目添加（或移除）

性质，以审计项目的插件属性。

```
protected void run() {
    Collection<IProject> projects = getSelectedProjects();
    for (IProject project : projects) {
        /*
         * The builder can be directly associated with a project
         * without the need for a builder...
         */
        if (false)
            toggleBuilder(project);
        // ... or more typically, the a nature can be used
        // to associate a project with a builder
        if (true)
            toggleNature(project);
    }
}

private void toggleNature(IProject project) {
    if (PropertiesAuditorNature.hasNature(project)) {
        PropertiesAuditorNature.removeNature(project);
    }
    else {
        PropertiesAuditorNature.addNature(project);
    }
}

private void toggleBuilder(final IProject project) {
    // If the project has the builder,
    // then remove the builder and all audit markers
    if (PropertiesFileAuditor.hasBuilder(project)) {
        PropertiesFileAuditor.deleteAuditMarkers(project);
        PropertiesFileAuditor.removeBuilderFromProject(project);
    }
    // otherwise add the builder
    // and schedule an audit of the files
    else {
        PropertiesFileAuditor.addBuilderToProject(project);
        new Job("Properties File Audit") {
            protected IStatus run(IProgressMonitor monitor) {
                try {
                    project.build(PropertiesFileAuditor.FULL_BUILD,
                        PropertiesFileAuditor.BUILDER_ID, null, monitor);
                }
                catch (CoreException e) {
                    FavoritesLog.logError(e);
                }
                return Status.OK_STATUS;
            }
        }.schedule();
    }
}

toggleNature方法在PropertiesAuditorNature中调用一些新方法以执行实际的项目性质的添加和移除。
public static void addNature(IProject project) {
    // Cannot modify closed projects.
    if (!project.isOpen())
```

```
        return;

    // Get the description.
    IProjectDescription description;
    try {
        description = project.getDescription();
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }

    // Determine if the project already has the nature.
    List<String> newIds = new ArrayList<String>();
    newIds.addAll(Arrays.asList(description.getNatureIds()));
    int index = newIds.indexOf(NATURE_ID);
    if (index != -1)
        return;

    // Add the nature
    newIds.add(NATURE_ID);
    description.setNatureIds(newIds.toArray(
        new String[newIds.size()]));

    // Save the description.
    try {
        project.setDescription(description, null);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
}

public static boolean hasNature(IProject project) {
    try {
        return project.isOpen() && project.hasNature(NATURE_ID);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return false;
    }
}

public static void removeNature(IProject project) {
    // Cannot modify closed projects.
    if (!project.isOpen())
        return;
    // Get the description.
    IProjectDescription description;
    try {
        description = project.getDescription();
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }
}
```

```
// Determine if the project has the nature.
List<String> newIds = new ArrayList<String>();
newIds.addAll(Arrays.asList(description.getNatureIds()));
int index = newIds.indexOf(NATURE_ID);
if (index == -1)
    return;

// Remove the nature
newIds.remove(index);
description.setNatureIds(newIds.toArray(
    new String[newIds.size()]));

// Save the description.
try {
    project.setDescription(description, null);
}
catch (CoreException e) {
    FavoritesLog.logError(e);
}
}
```

14.4 RFRS相关事项

《RFRS Requirements》的“构建”一节包括6个（3个要求和3个最佳做法）与构建器相关的内容。

14.4.1 使用构建器以转换资源（RFRS 3.8.1）

要求#1说明：

任意将资源从一种格式转换至另一种资源同步的格式（如编译器）的扩展项，必须使用build API和org.eclipse.core.resources.builders扩展点。

为了通过该要求，在开始时展示你的构建器在运行。描述它是如何被触发的和它转换哪些资源。关闭General > Workspace首选项页的Build automatically首选项，并展示你的构建器没有运行。触发Project > Rebuild Project命令以展示你的构建器正确地处理了所有的累积的更改。

14.4.2 不要替代已有构建器（RFRS 3.8.3）

要求#2说明：

扩展项不可以替代与由工作台或其他提供者提供的项目性质关联的构建器。

配置项目以使用你的构建器。打开项目的.project文件以展示你的构建器已经被添加并且已有构建器，如org.eclipse.jdt.core.javabuilder，没有一个已经被移除。

14.4.3 不要滥用术语“构建”（RFRS 5.3.8.1）

最佳做法#3说明：

术语“build”不应被过多使用，使得它具有超出使用Eclipse build API触发的构建过程的含义。也就是说，不要在你的产品实现或文档中使用术语“构建”以描述一个不是作为构建器在工作台中实现的过程。

展示你产品文档中所有使用术语“build”的位置，并确认所有这些使用与你插件的构建器相关。

14.4.4 标记已创建的资源为“派生的”(RFRS 5.3.8.2)

最佳做法#4说明:

在资源不是源代码(.java文件中),或一些可以由用户更改或将不会由用户更改的其他类型的手动项,或部署至一个运行时平台所必需的时,由构建器创建的资源应被标识为派生的。

对于这项测试,阐述由你的构建器创建的所有资源被标记为派生。在由Java编译器创建的.class文件的情况下,你将需要打开属性对话框并展示Derived选项已经被选中(图14-14)。

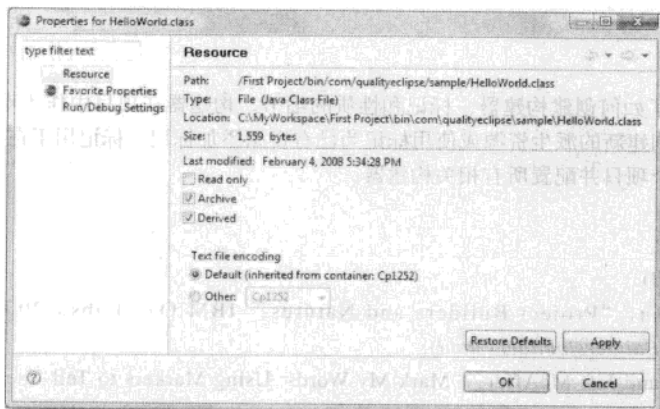


图14-14 HelloWorld.class文件的属性对话框

14.4.5 响应清理构建请求(RFRS 5.3.8.3)

最佳做法#5说明:

构建器响应CLEAN_BUILD请求。CLEAN_BUILD请求构建器忘记所有由它使用clean()方法的覆盖版本私下维护的附加状态。一个用户触发的清理构建随后为一个FULL_BUILD请求。

对于这项测试,检查构建器存在问题时的输出,并使用属性视图记录它们的时间戳。触发一个CLEAN_BUILD请求(Projects > Clean...).核对接受测试的构建器的属性已经被重新创建(检查时间戳)了,并且构建器运行以没有错误地完成(检查错误日志)。

14.4.6 在可能时使用IResourceProxy(RFRS 5.3.8.4)

最佳做法#6说明:

构建器在请求了一个IncrementalProjectBuilder.FULL_BUILD时,常常需要处理项目中的所有资源。从Eclipse 2.1开始有一个可用的改进技术。应使用IResourceProxyVisitor取代IResourceVisitor。代理访问者提供了对于轻量级的IResourceProxy对象的条目。这些可以在需要时返回一个真正的IResource对象。但当不需要时,它们可以为所有构建提高改进的构建器性能。

为了通过这项测试,你需要防止使用IResourceVisitor对象并使用IResourceProxyVisitor对象作为替代。搜索你的插件源代码以查找对于IResourceVisitor的引用并解释为什么不能使用IResourceProxyVisitor作为替代。

14.4.7 构建器必须由性质添加 (RFRS 5.3.8.5)

最佳做法#7说明:

构建器必须由性质添加至项目。性质实现, 如同org.eclipse.core.resources.natures扩展项中标识的那样, 将添加所有必需的构建器作为configure()方法的一部分。

对于该测试, 展示你的plugin.xml文件中的性质定义。使用你的性质创建一个项目, 并说明构建器被自动配置。添加你的性质至一个已有项目的.project文件并展示你的构建器在文件被保存后立即被触发。

14.5 总结

本章深入探讨了如何创建构建器、标记和性质的细节。构建器在项目中作为对资源更改的响应执行。构建器可以创建新的派生资源或使用标记为已有资源添加标记。标记用于在资源内标记位置, 而性质用于标记整个项目并配置所有相关构建器。

参考文献

本章资源 (2.9节).

Arthorne, John, "Project Builders and Natures," IBM OTI Labs, 2003 (www.eclipse.org/articles/Article-Builders/builders.html).

Glozic, Dejan and Jeff McAffer, "Mark My Words: Using Markers to Tell Users about Problems and Tasks," IBM OTI Labs, 2001 (www.eclipse.org/articles/Article-Mark%20My%20Words/Mark%20My%20Words.html).



第15章 实现帮助

无论你的插件可能是多么好和易于使用的，最终，用户都将有一些关于某些功能是如何工作的疑问，或者他们可能需要了解一些操作或设置的进一步细节。幸运的是，Eclipse提供了一个全面的框架以用于在程序内包含联机帮助。

本章首先一个关于如何访问Eclipse帮助系统的概述，然后是一个关于如何为你的程序实现帮助的讨论。接着对一个关于如何添加环境相关（F1）的帮助进行讨论。然后用图示说明如何在程序中访问帮助系统。最后是一个关于如何引导用户使用备忘录在任务序列中跳转的讨论。

15.1 使用帮助

用户可以通过使用Help > Help Contents菜单访问你的完整产品文档（图15-1）。该菜单将打开Eclipse Help窗口（图15-2）。Help窗口独立于Eclipse工作台窗口，以使在两者之间切换更为容易。

在Eclipse的早期版本中，Help窗口一开始是作为它自己的透视图实现。这使得在使用不同工作台功能（尤其是模态对话框或向导）时访问帮助系统是十分困难的。

提示 如果你正创建一个Eclipse RCP，你可以使用一个Help Contents...菜单项添加帮助菜单。要了解更多内容，参见<http://downloads.instantiations.com/HelpComposerDoc/integration/latest/docs/html/gettingstarted/linkhelptomenu.html>。

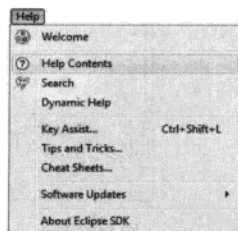


图15-1 Eclipse帮助菜单

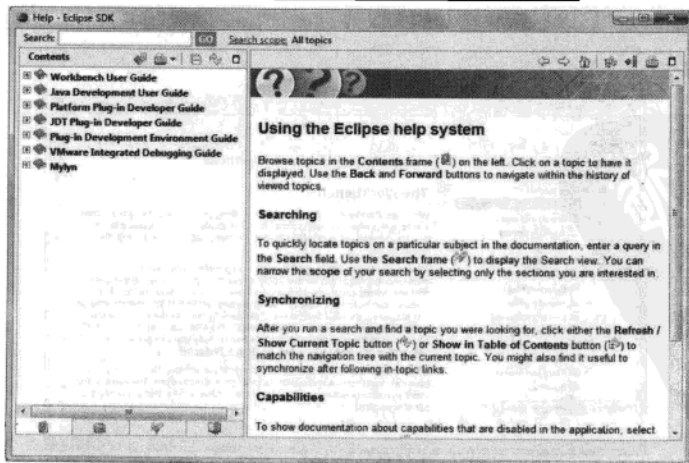


图15-2 Eclipse帮助窗口

Help窗口具有几个主要的组件。左上角是一个搜索字段。输入一个搜索词语并点击Go按钮将使帮助系统搜索系统中所有的帮助主题。为搜索词语添加引号将使用精确匹配搜索而不是默认的模糊搜索。模糊搜索使用词根并且比精确匹配搜索提供更多的结果。但当搜索精确词语或短语时效果不是很好。

在Search字段的右侧是一个链接。该链接用于设置搜索范围。点击Search scope链接将打开选择搜索范围对话框（图15-3）。

在该对话框内，你可以选择搜索所有可用主题或定义为一个工作组的主题集。点击对话框中的New...按钮将允许你创建一个由最高级帮助主题组成的新的工作集（图15-4）。在这两个对话框中点击OK按钮将返回至帮助窗口。

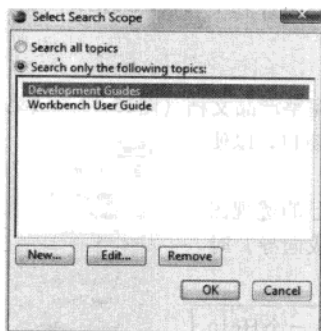


图15-3 选择搜索范围对话框

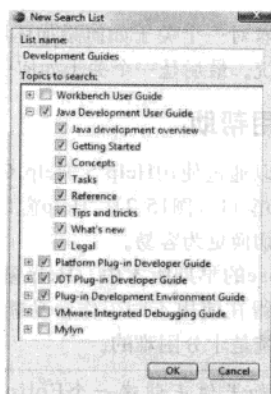


图15-4 新建搜索列表对话框

在搜索字段的下面是一个系统中可用的最高级帮助文档的列表。在这里你可以找到内建的帮助文档，如《Workbench User Guide》和《Platform Plug-in Developer Guide》，和由其他插件添加的文档。每一个文档都可以扩展以显示它自身的帮助主题。选择一个帮助主题将在主题列表右侧内容区域显示帮助页（图15-5）。

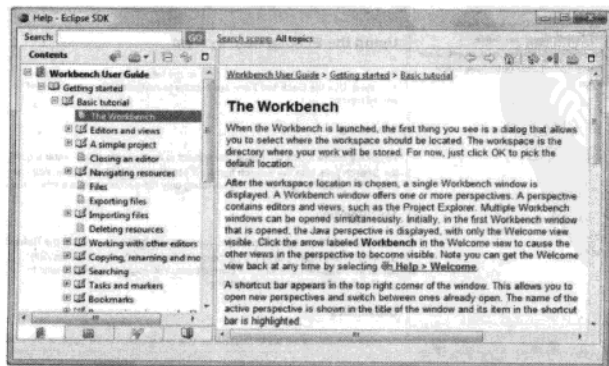


图15-5 显示选中帮助主题的帮助窗口

15.2 实现帮助

Eclipse提供将你自己的帮助文本添加至帮助环境所必需的基础结构。Eclipse不关心帮助文件是如何创建的,并乐于使用HTML或PDF格式的内容(如果你愿意,你甚至可以使用XHTML创建动态的、“活动的”帮助)。你的帮助既可以集成至主插件,也可以放置于它自己的独立插件(这是Eclipse基本插件的普遍做法)。

在完成创建你的帮助内容之后。此刻,我们假设是超文本标记语言(HTML),将它们集成至Eclipse包括四个简单步骤:

- 1) 创建一个新插件项目以容纳帮助插件。
- 2) 在项目内设置帮助文件目录结构。
- 3) 更新内容表(toc)文件以引用帮助文件。
- 4) 更新插件清单以引用toc文件。

15.2.1 新建帮助项目

基于Eclipse项目创建向导的功能,设置你的帮助项目所需的大部分的文件可以自动生成。你将从使用New Project向导创建一个新的Plug-in Project开始(图15-6)。创建项目包含相当多的步骤,但考虑到有一部分项目结构将被自动创建,这些步骤是值得的。

在New Plug-in Project向导的第一页(图15-7),输入“com.qualityeclipse.favorites.help”作为项目名称。让Use default单选框保持选中以使项目创建于默认工作区。最后,当你不想要创建任意Java文件,取消选中Create a Java project选项。

在向导的第二页(图15-8),让插件ID保持为“com.qualityeclipse.favorites.help”,并更改名称为“Favorites Help”。当你取消选中Java项目选项并不打算执行任意Java代码,没有必要使用插件启动类。

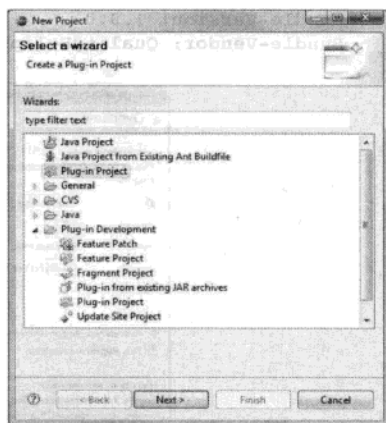


图15-6 新建项目向导

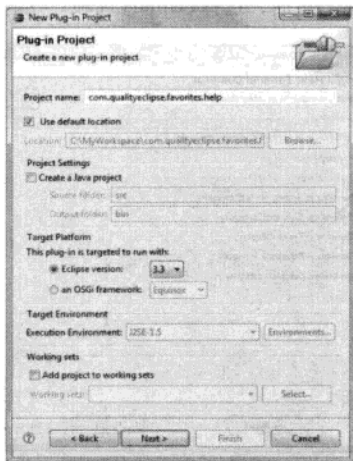


图15-7 新建插件项目向导

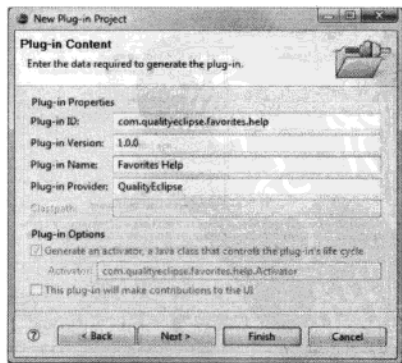


图15-8 插件内容页

点击Finish按钮以创建该项目。初始的项目配置将包含一些文件，包括初始的项目清单文件MANIFEST.MF。该文件自动在清单文件编辑器中打开（图15-9）。如果你切换至编辑器的MANIFEST.MF页，你将看到以下内容：

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Favorites Help
Bundle-SymbolicName: com.qualityeclipse.favorites.help
Bundle-Version: 1.0.0
Bundle-Vendor: QualityEclipse
```

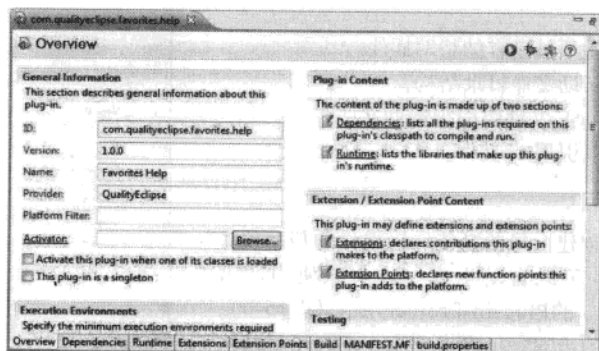


图15-9 插件清单文件概述页

在此时，你有了一个不具有任意帮助内容的空项目。切换至清单编辑器的Extensions页并点击Add...按钮以打开New Extension向导。切换至Extensions Wizards选项卡并在右边的列表中选择帮助内容（图15-10）。

在向导的下一页（图15-11），设置Label for table of contents字段为“Favorites Guide”。这是将作为一个最高级项出现于Help窗口的内容区域的文档的名称。然后，选择Primary选项。这将创建一

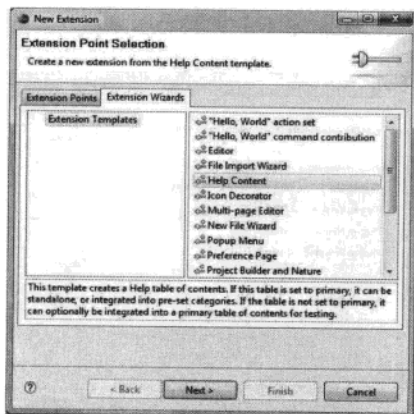


图15-10 扩展项向导页

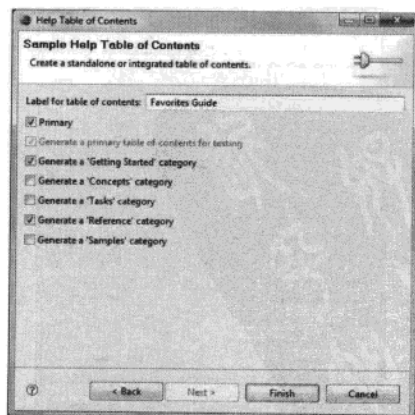


图15-11 内容页的帮助表

个主toc文件。主toc文件表示Help窗口的最高级项。项目中可以存在任意数量的toc文件，但它们将全部为一个或多个主toc文件的下级文件。

最后，你将可以选择在帮助文档内部，创建一个或多个包含主要主题区域的预定义二级toc文件。创建二级toc文件是可选的。所有你的帮助文件可以从你的主toc文件较容易地引用。使用多个二级toc文件分隔帮助提供了更好的粒度划分，并且使得几个人不冲突地同时更新文档的不同部分变得更加容易。在这里，选择创建Getting Started和Reference类别的选项以阐述不同的文件是如何组织的。点击Finish以完成该过程并生成所有的文件。

15.2.2 插件清单文件

基于稍早选中的选项，将创建多个文件（图15-12），包括几个伪HTML文件和以下XML文件：plugin.xml、toc.xml、tocgettingstarted.xml和tocreference.xml。

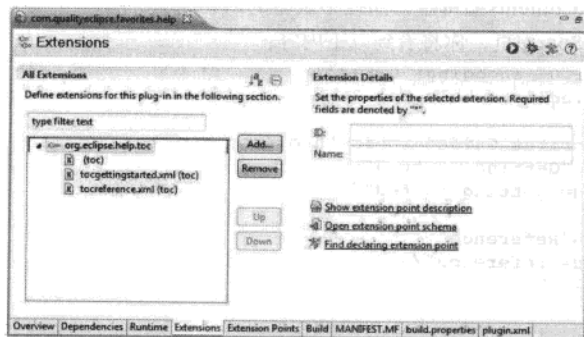


图15-12 清单编辑器的扩展项页

如果你切换至编辑器的plugin.xml页，你将看到以下内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
    point="org.eclipse.help.toc">
    <toc
      file="toc.xml"
      primary="true"/>
    <toc file="tocgettingstarted.xml"/>
    <toc file="tocreference.xml"/>
  </extension>
</plugin>
```

这里最重要的是org.eclipse.help.toc扩展点的扩展项。该扩展点用于指定主toc文件和二级toc文件。file属性用于指定将被使用的toc文件，而primary属性表示一个独立的toc文件是否应作为一个最高级文档出现于帮助窗口的帮助主题列表中。任意primary属性不为true的toc文件将直到它们从一个或多个主toc文件链接才会显示。

org.eclipse.help.toc扩展点还定义了一个附加的、很少用到的extradir属性。它指定包含附件文档文件的目录名。除非它们由一个toc文件中的特定主题元素引用，这些文件不可以通过主题列表访问，

但可以被索引并通过帮助搜索功能访问。

提示 如果你在创建一个Eclipse RCP, 你可以替换默认的Eclipse帮助主页面。为了了解更多内容, 参见

http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/ua_help_setup_preferences.htm

和

<http://dev.eclipse.org/mhonarc/lists/platform-help-dev/msg00844.html>.

15.2.3 内容表 (toc) 文件

然后, 让我们看一看生成的toc文件。toc.xml文件代表插件的主帮助主题条目。Eclipse 3.4 添加了一个专门的Table of Contents编辑器, 而较早版本的Eclipse提供了一个简单的层次结构视图 (图15-13)。如果你切换至Source页, 你将看到以下内容:

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Favorites Guide" topic="html/toc.html">
  <topic label="Getting Started">
    <anchor id="gettingstarted"/>
  </topic>
  <topic label="Reference">
    <anchor id="reference"/>
  </topic>
</toc>
```

在toc文件内部, label属性指定了在主题列表中显示的文本, 而topic属性指定了一个链接链接至一个文本页。该文本页应在主题被选中时显示。

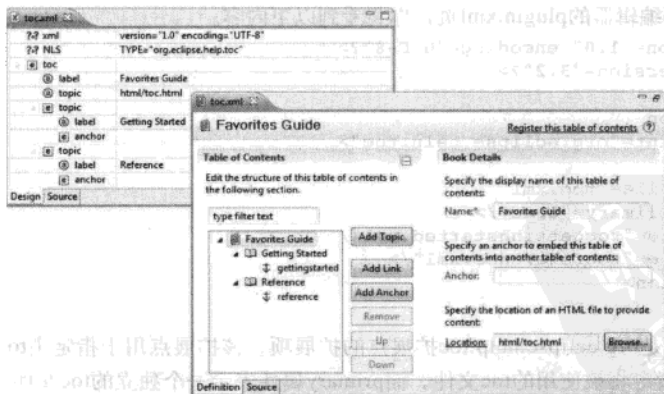


图15-13 内容表编辑器Eclipse 3.3版本在左侧, 而Eclipse 3.4版本在右侧

该toc文件的结构说明了如何从多个、嵌套的toc文件创建帮助主题树。Eclipse支持两种不同的方法以创建帮助主题树: 由顶向下嵌套和从底向上组合。上述的toc文件说明了第二种方法。在从底向上组合中, toc文件将定义不同的锚点 (anchor point)。其他toc文件可以向这些锚点添加附加主题。

在这里，已经为文档定义了两个子主题（“Getting Started”和“Reference”），每一个都定义了一个锚点（本质上是一个让其他toc文件填充的空容器）。请注意这些子主题的每个都可能也已经定义了它自己的硬编码的文档链接（你将在下一个你查看的toc文件中看到一个这样的示例）。

然后，查看两个剩下的toc文件的其中之一（两个都是同样的样式，所以你只需要查看它们其中之一）。tocgettingstarted.xml文件与以下内容类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Getting Started" link_to="toc.xml#gettingstarted">
  <topic label="Main Topic"
    href="html/gettingstarted/maintopic.html">
    <topic label="Sub Topic"
      href="html/gettingstarted/subtopic.html" />
    </topic>
  <topic label="Main Topic 2">
    <topic label="Sub Topic 2"
      href="html/gettingstarted/subtopic2.html" />
    </topic>
  </toc>
```

该toc文件提供了至使用href属性表示插件文档的实际HTML文件的主题链接（你将在稍后替代由具有链接至文档文件的向导所生成的样本文件）。该toc文件最有趣的部分是link_to属性的使用。toc文件使用该属性以将它自身链接至主toc文件的gettingstarted锚。

提示 请注意你的插件标准页面不限制于仅链接至你自己的锚点。核心Eclipse文档提供了许多锚点以让你可以附加帮助文件。如果你的插件扩展了基本Eclipse功能之一，将你的帮助页面链接至Eclipse文档的恰当章节是很好的选择。

我们刚讨论过的toc文件都提供了从底向上组织的示例。在这种方法中，二级toc文件链接至主toc文件，而主toc文件对二级toc文件一无所知。这种方法紧密地反映了Eclipse内使用的扩展点概念。相反的从上至下嵌套将这种方法反过来，以使主toc文件直接链接至二级toc文件。

转换这两个toc文件以使用从上至下的方法是比较简单的。首先，使用指向二级toc文件的link属性替换主toc文件中的anchor属性。toc.xml文件将与以下内容类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Favorites Guide" topic="html/toc.html">
  <topic label="Getting Started">
    <link toc="tocgettingstarted.xml" />
  </topic>
  <topic label="Reference">
    <link toc="tocreference.xml" />
  </topic>
</toc>
```

然后，你可以从二级toc文件中移除link_to属性，因为不再需要它们了。

提示 从上至下方法提供了对帮助文档中包含文件的可见性的更好控制，并且没有为其他实体提供扩展由该插件提供的帮助的任何机会。相反地，从底向上方法在结构化帮助文档方面，和随着时间使用从插件内部和外部获取的附加添加项有机地增加帮助文档的内容方面，提供了很好的灵活性。

国际化 如果你的程序需要支持多种语言，你的toc文件和文档文件可能被转换为多种语言，并被放置于你的插件根目录的特殊命令的子目录中（为了了解更多信息，参见16.3.3节）。转换后的文件应被放置nl/<language>于或nl/<language>/<country>目录，在这两个目录中，<language>和<country>代表用于表示目标语言和国家的双字母代码。比如，巴西翻译版本将会放置于nl/pt/br目录，而标准葡萄牙翻译版本将会放置于nl/pt目录。帮助系统将首先查看nl/<language>/<country>目录。如果没发现任何内容，将使用nl/<language>作为替代。如果没有发现目标语言的翻译版本，将会默认使用插件根目录中的文件。

15.2.4 创建HTML内容

除了XML文件之外，向导还创建了几个类HTML文件。

```
html/
  gettingstarted/
    maintopic.html
    subtopic.html
    subtopic2.html
  reference/
    maintopic.html
    subtopic.html
    subtopic2.html
  toc.html
```

假设你使用以下结构为Favorites视图创建你自己的文档文件：

```
html/
  gettingstarted/
    installation.html
    favorites_view.html
    adding_favorites.html
    removing_favorites.html
  reference/
    view.html
    actions.html
    preferences.html
  toc.html
```

那么你将需要更新这两个toc文件。比如，tocgettingstarted.xml文件，将最终与以下内容类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Getting Started">
  <topic label="Installation"
    href="html/gettingstarted/installation.html"/>
  <topic label="Favorites View"
    href="html/gettingstarted/favorites_view.html">
  <topic label="Adding Favorites"
    href="html/gettingstarted/adding_favorites.html" />
  <topic label="Removing Favorites"
    href="html/gettingstarted/removing_favorites.html" />
  </topic>
</toc>
```

提示 如果你的程序需要使用较多的帮助文件，并且你将把你的插件作为一个目录而不是JAR文件分发，那么你可以考虑将所有文件放置于一个ZIP文件中。如果你打包你的帮助文件为一个名为doc.zip的文件（具有同样的目录结构），Eclipse将可以发现它们。Eclipse将在从插件目录中查找它们之前，在doc.zip文件中查找帮助文件。

包括这些更改在内，帮助插件的功能就完成了。将该新的帮助插件放置到位并启动Eclipse将添加Favorites Guide至Help窗口的文档列表（图15-14）。

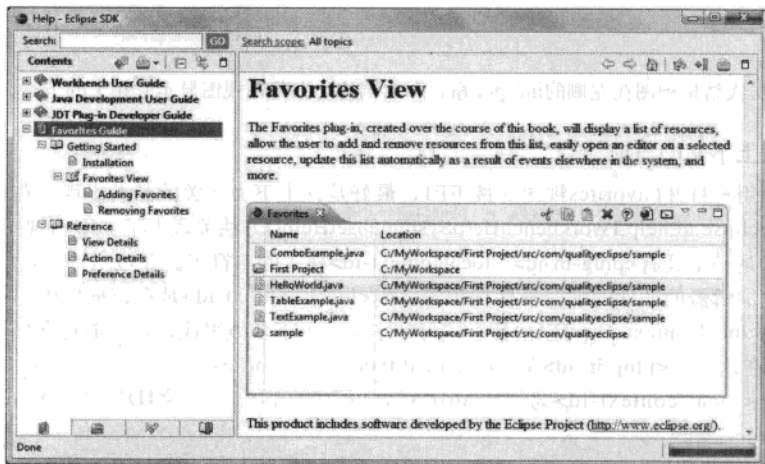


图15-14 显示收藏夹指南的帮助窗口

动态帮助内容 在Eclipse 3.2及以后版本，帮助系统支持使用XHTML的动态帮助组合。帮助页面可以使用Eclipse特定的标记以提供基于os/ws/arch值、可用功能、插件存在等的过滤。内容也可以通过使用新的<include>标记在多个文档间重用。这些功能允许帮助内容以在查看时匹配上下文环境。为了了解更多关于在Eclipse中使用XHTML的信息，参见：www.eclipse.org/eclipse/platform-ua/proposals/help/dynamic_content/HelpDynamicContent.html。

15.3 上下文相关的帮助 (F1)

Eclipse为窗口小部件、窗口、操作和菜单提供了使用F1键来使用上下文相关的帮助。该标准将出现在一个浮动的“infopop”窗口或基于用户的Help首选项的动态Help视图内。所出现的上下文相关帮助可以包含少量关于选中项的帮助和至更详细文档的链接。比如，打开Eclipse Hierarchy视图并按下F1将显示包含Hierarchy视图的相关链接和总体视图的上下文相关的帮助（图15-15）。

为了添加上下文相关的帮助至一个项，你需要关联上下文ID和该项，提供该项的一个简短描述，并创建一个与该项相关的链接列表。

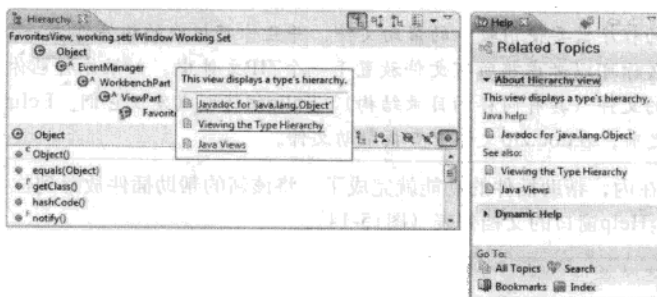


图15-15 层次结构视图在左侧的infopop窗口和右侧的动态帮助视图显示它相关的上下文相关帮助

15.3.1 关联上下文ID与项

无论何时用户打开Favorites视图并按下F1，最好是让上下文相关的帮助出现。首先，你需要使用来源于org.eclipse.ui.help.IWorkbenchHelpSystem的setHelp()方法关联上下文ID与视图。

上下文ID是一个具有<plug-in-id>.<localcontext-id>样式的字符串。在这里，<plug-in-id>是定义工作台视图或编辑器的插件的唯一插件标识符，而<local-context-id>是在该插件内部标识上下文的唯一标识符。<localcontext-id>可以由任意字母数字和一个下划线组成，但不能包含任何空格或英文句号（“.”）。在这里，<plug-in-id>是“com.qualityeclipse.favorites”，并且你已经为你的Favorites视图选择了一个<local-context-id>为“favorites_view”。所以，上下文ID为“com.qualityeclipse.favorites.favorites_view”。

由于Favorites视图由一个表部件组成，你将需要关联帮助上下文ID与表部件本身。我们首先在FavoritesView类中创建一个与以下内容类似的setHelpContextIDs()方法：

```
// FavoritesView.java
private void setHelpContextIDs() {
    IWorkbenchHelpSystem helpSystem =
        getSite().getWorkbenchWindow().getWorkbench().getHelpSystem();

    helpSystem.setHelp(
        viewer.getControl(),
        "com.qualityeclipse.favorites.favorites_view");
}
```

然后，你可以从FavoritesView类中的已有的createPartControl()方法调用该方法。

帮助上下文由子控件继承。在这里，一个帮助上下文被分配至Favorites查看器控件。该控件没有后代。如果你将要分配一个帮助上下文至一个复合组件，那么它的后代将继承同样的帮助上下文，除非你通过在子控件上调用setHelp()覆盖它。

Favorites视图还定义了几个不同的方法。你可以使用这些方法以关联帮助上下文ID。通过改进setHelpContextIDs()方法完成该任务以关联帮助与操作：

```
helpSystem.setHelp(filterAction,
    "com.qualityeclipse.favorites.filter");
```

帮助还可以被关联至菜单项。通过添加以下内容改进RemoveFavorites ContributionItem.fill(Menu,int)：

```
IWorkbenchHelpSystem helpSystem =
    viewSite.getWorkbenchWindow().getWorkbench().getHelpSystem();
helpSystem.setHelp(menuItem,
    "com.qualityeclipse.favorites.remove");
```

帮助上下文ID还可以通过定义helpContextId属性被分配至命令、菜单添加项和插件清单文件中定义的处理程序。比如，你可以像以下所示的那样改进“Open Favorites View”命令的定义：

```
<extension point="org.eclipse.ui.commands">
    <command
        categoryId="com.qualityeclipse.favorites.commands.category"
        description="Open the Favorites view"
        helpContextId="com.qualityeclipse.favorites.favorites_view"
        id="com.qualityeclipse.favorites.commands.openView"
        name="Open Favorites View">
    </command>
```

请注意如果插件清单中使用了本地上下文标识符，那么用于声明插件的唯一标识符将被添加前缀以组成完整的上下文标识符。

15.3.2 IWorkbenchHelpSystem API

IWorkbenchHelpSystem接口定义了一些常用API。这些API可以分配帮助上下文ID并在程序中显示帮助，例如：

- displayContext(IContext context, int x, int y)——显示给定上下文的上下文相关帮助。
- displayDynamicHelp()——显示当前UI上下文的动态帮助。
- displayHelp()——显示整个帮助主题。
- displayHelp(IContext context)——显示给定上下文的上下文相关标准。
- displayHelp(String contextId)——调用帮助支持系统以显示给定的帮助上下文ID。
- displayHelpResource(String href)——显示具有给定URL的帮助资源的帮助内容。
- displaySearch()——显示帮助搜索系统。
- hasHelpUI()——返回是否安装了UI帮助系统。
- isContextHelpDisplayed()——返回上下文相关帮助是否正显示。
- search(String expression)——使用帮助搜索系统开始搜索。
- setHelp(Control control, String contextId)——设置给定控件的给定帮助上下文ID。
- setHelp(IAction action, String contextId)——设置给定操作的给定帮助上下文ID。
- setHelp(MenuItem item, String contextId)——设置给定菜单项的给定帮助上下文ID。
- setHelp(Menu menu, String contextId)——设置给定菜单的给定帮助上下文ID。

15.3.3 创建上下文相关的帮助内容

当上下文ID已经被分配给视图时，你需要为每一个由描述和一个链接集组成的帮助项创建内容。在一个或多个XML格式的上下文清单文件中描述了这些内容。对于在上一节分配了上下文ID的项来说，contexts.xml文件可能与以下内容类似：

```
<contexts>
    <context id="favorites_view">
        <description>This is the Favorites view.</description>
        <topic href="html/gettingstarted/favorites_view.html"
            label="Using the Favorites View"/>
    </context>
</contexts>
```

```

<topic href="html/gettingstarted/installation.html"
      label="Installing the Favorites View"/>
<topic href="html/reference/preferences.html"
      label="Favorites View Preferences"/>
</context>
<context id="filter">
  <description>Filter the contents
                    of the Favorites view</description>
  <topic href="html/reference/filter.html"
        label="Using the Favorites view Filter"/>
</context>
<context id="remove">
  <description>Remove an element
                    from the Favorites view</description>
  <topic href="html/gettingstarted/removing_favorites.html"
        label="Removing elements from the Favorites view"/>
</context>
</contexts>

```

在contexts.xml文件内部，每一个上下文ID都由它自己的上下文元素进行描述。每一个上下文元素具有一个描述元素和零个或多个链接至实际文档的主题元素。每一个主题元素具有一个提供链接的href属性和一个描述链接的文本的label属性。它将出现于上下文相关的帮助中。

15.3.4 上下文扩展点

一旦上下文ID已经关联至视图和操作，并且已经定义每一个帮助项的上下文了，你需要更新你的plugin.xml文件以指向contexts.xml文件并将它关联至主插件。内建的Eclipse向导使得这项工作更为容易。

我们首先在帮助项目中打开plugin.xml文件并切换至扩展项页（图15-16）。然后，点击Add...按钮。

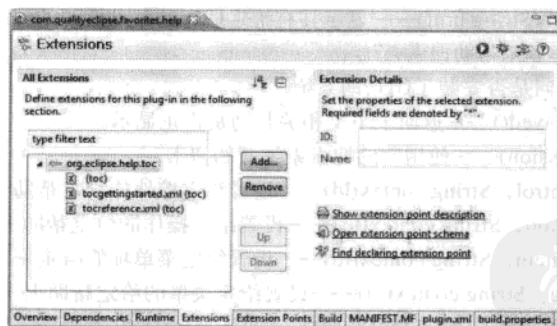


图15-16 显示扩展项页的收藏夹帮助清单文件

当打开New Extension向导时，从所有可用扩展点列表中选择org.eclipse.help.contexts（图15-17）。如果你在列表中没有发现org.eclipse.help.contexts，取消选中Show only extension points from the required plug-ins单选框。点击Finish按钮以将该扩展项添加至插件清单。

现在，让我们回到插件清单编辑器的Extensions页，右键点击org.eclipse.help.contexts扩展项并选择New > contexts。这将立即添加一个上下文项至插件清单。点击该新的上下文项将显示属性，以使它们可以根据如下内容进行修改（图15-18）：

- file——“contexts.xml”
上下文XML文件的名称。
- plugin——“com.qualityeclipse.favorites”
与透视图关联的文本标签。

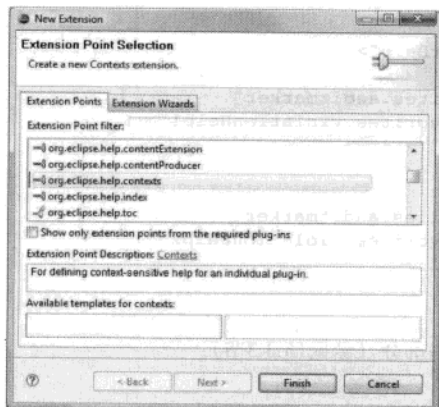


图15-17 显示选中org.eclipse.help.contexts
扩展点的新建扩展项向导

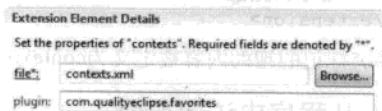


图15-18 显示收藏夹上下文文件属性
的扩展项元素细节

如果你切换至插件清单编辑器的plugin.xml页，你将看到定义新的上下文文件的XML新内容：

```
<extension
  point="org.eclipse.help.contexts">
  <contexts
    file="contexts.xml"
    plugin="com.qualityeclipse.favorites">
  </contexts>
</extension>
```

plugin属性对于关联该上下文文件与com.qualityeclipse.favorites插件是很重要的。如果没有指定该内容，上下文文件将被关联至定义它的本地插件。

请注意，不同插件的多个上下文文件可以被关联至上下文ID。这将允许一个插件扩展由另一个插件提供的上下文帮助。

现在你已经完成了上下文相关的帮助的定义，你可以通过打开Favorites视图并按下F1键对它进行测试。“favorites_view”上下文ID的帮助内容应出现于Help视图中（图15-19）或浮动的infopop窗口，要显示菜单项的帮助，在菜单项高亮显示时，按下F1键。

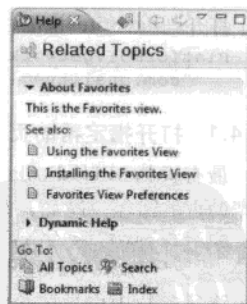


图15-19 显示收藏夹视图的
上下文相关帮助的帮助视图

动态帮助上下文 参见IContextProvider以了解关于在程序中关联帮助上下文与视图和编辑器的内容。

15.3.5 标记帮助

org.eclipse.ui.ide.markerHelp扩展点允许插件关联帮助上下文ID与特定的标记类型。第14章有一个表示两个不同冲突的标记类型，所以你需要进一步验证帮助声明，为每一种冲突类型创建一个声明。表达式<attribute name="violation" value="1"/>表示帮助上下文应仅应用于具有值为“1”的violation属性的标记。

```
<extension point="org.eclipse.ui.ide.markerHelp">
  <markerHelp
    markerType="com.qualityeclipse.favorites.auditmarker"
    helpContextId="com.qualityeclipse.favorites.violationHelp1">
    <attribute name="violation" value="1"/>
  </markerHelp>
  <markerHelp
    markerType="com.qualityeclipse.favorites.auditmarker"
    helpContextId="com.qualityeclipse.favorites.violationHelp2">
    <attribute name="violation" value="2"/>
  </markerHelp>
</extension>
```

冲突标记的帮助内容被定义为contexts.xml文件的一部分（参见15.3.3节）。

15.4 从程序中访问帮助

到目前为止，你已经了解了如何集成帮助至Eclipse帮助窗口和使用标准Eclipse机制，如Help > Help Contents菜单和F1键，访问它。可能有时你需要用除标准机制之外的其他方法提供帮助。

如同稍早所见的那样，IWorkbenchHelpSystem接口定义了大量常用API。这一节关注display方法的一部分。

为了在程序中地打开Help窗口，不使用参数调用displayHelp()方法。为了在程序中打开具有指定上下文ID的上下文相关的帮助，使用上下文ID字符串作为唯一参数调用displayHelp()方法。比如，为了打开与Favorites视图相关的上下文相关的帮助，使用以下代码。

```
PlatformUI.getWorkbench().getHelpSystem().displayHelp(
    "com.qualityeclipse.favorites.favorites_view");
```

15.4.1 打开指定帮助页

最有趣的API是displayHelpResource()方法。该方法接收一个字符串参数。该参数表示将要显示的帮助页的路径。比如，为了打开Favorites插件的主帮助页，使用以下代码：

```
PlatformUI.getWorkbench().getHelpSystem().displayHelpResource(
    "/com.qualityeclipse.favorites.help/html/toc.html");
```

路径参数由包含该帮助文件的插件的ID和相对于插件根目录的资源的路径组成。基于上一个示例，你可以轻松地通过在插件清单中定义一个工具栏按钮以添加一个自定义帮助按钮至Favorites视图（图15-20）的工具栏（参见6.2.6节），并使用以下方法创建一个处理器。

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    // Show the help window
    // PlatformUI.getWorkbench().getHelpSystem().displayHelp();
    // Show context sensitive help
    // PlatformUI.getWorkbench().getHelpSystem().displayHelp(
    //     "com.qualityeclipse.favorites.favorites_view");
```

```
// Show a specific help page
PlatformUI.getWorkbench().getHelpSystem().displayHelpResource(
    "/com.qualityeclipse.favorites.help/html/toc.html");
return null;
}
```

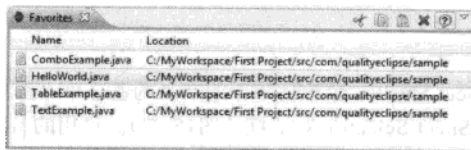


图15-20 显示新建的帮助按钮的收藏夹视图

15.4.2 打开网页

除了在Eclipse Help窗口中打开指定帮助页之外，你可能还想要使用浏览器打开一个指定网页。Eclipse包含一个类org.eclipse.swt.program.Program。该类用于启动外部程序，包括系统的网络浏览器。你应尤其关注该类中的launch()方法。该方法接收一个字符串作为参数。该字符串包含至将要启动的程序的路径，或将要访问的网页的URL。

你现在可以添加一个按钮至Favorites视图的工具栏（图15-21）。该工具栏将打开一个指定的网页（在本书中使用该网页作为示例）。通过在插件清单中定义一个工具栏按钮以完成该工作（参见6.2.6节）。该插件清单拥有一个具有以下方法的OpenWebPageHandler处理器。

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {

    // Show a page in an external web browser
    Program.launch("http://www.qualityeclipse.com");

    return null;
}
```

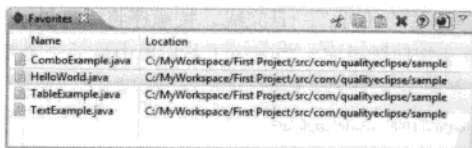


图15-21 显示新建Web按钮的收藏夹视图

这项技术在Windows中运行良好，但可能在其他平台，如Linux，就无法工作。要了解一个将在任意平台都能工作的替代方法，参见21.4节。

提示 你可以使用这项技术从你的程序生成一个E-mail消息至你的销售或支持部门。比如，执行以下代码：

```
Program.launch(
    "mailto:info@qualityeclipse.com" +
    "?Subject=Information Request")
```

将生成一个具有Information Request主题的E-mail消息。嵌入一个“?Body=”标记使你可以使用诸如用户的Eclipse配置之类的信息预填充消息主体。

15.5 备忘单

除了在Eclipse帮助系统中提供的静态帮助和由动态Help视图和infopop插件提供的上下文相关的帮助之外，Eclipse还包含了备忘单（Cheat Sheets）。它设计为向你展示一些完成任务的步骤，并自动启动任意必需的工具。

15.5.1 使用备忘单

用户可以通过Help > Cheat Sheets...菜单访问你的产品的备忘单（图15-22）。

这将打开Eclipse Cheat Sheet Selection对话框（图15-23）。可用的备忘单根据类别进行分组。双击一个备忘单或选择一个并点击OK按钮。

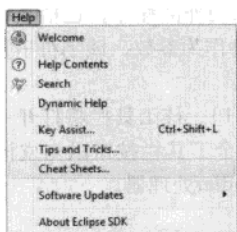


图15-22 Eclipse备忘单菜单项

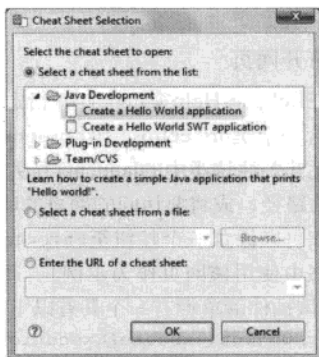


图15-23 备忘单选择对话框

选中的备忘单将打开为一个视图（图15-24）。同一时间仅只有一个备忘单可以是活动的，因此任意当前打开的备忘单首先将被关闭。每一个备忘单首先一个介绍，随后是一些步骤。在阅读了介绍之后，你可以提供点击Click to Begin按钮开始使用备忘单。这将展开并高亮显示下一步。使用Click to Perform按钮以执行与该步骤相关的任务。一旦完成，按顺序将高亮显示下一步。如果一个步骤是可选的，你可以使用Click to Skip按钮跳过它。当完成了备忘单中最后一个步骤时，它将自动重新启动。

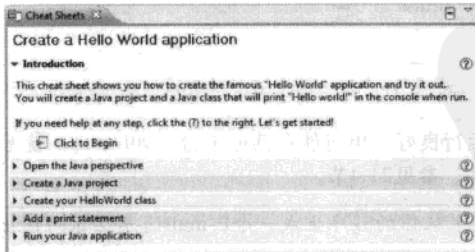


图15-24 备忘单示例

15.5.2 创建简单备忘单

创建备忘单包含几个步骤，而Eclipse提供了一个向导可以很快上手。在我们的示例中，我们将

放置新的备忘单于一个名为“cheatsheets”的文件夹中，所以我们首先选择File > New > Folder并使用打开的向导以创建新的文件夹。然后，选择File > New > Other...以打开新建向导并在User Assistance组中选择Cheat Sheet（图15-25）。在向导的下一页，选择“cheatsheets”文件夹，然后输入“FavoritesCheatSheet.xml”作为新的备忘单文件的名称。

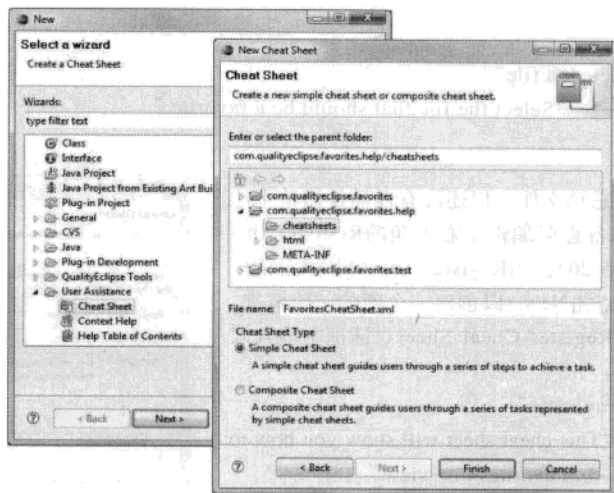


图15-25 新建备忘单向导

在你点击Finish之后，将打开备忘单编辑器（图15-26）。在左侧的树中选择步骤将在右侧显示该步骤的细节。

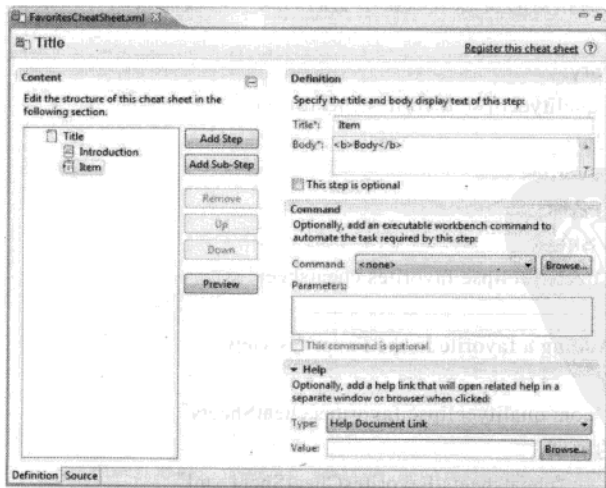


图15-26 备忘单编辑器

使用备忘单编辑器（图15-26），在左侧选择Title，并在右侧的标题字段中选择“Adding a favorite to the Favorites view”。继续这一过程，在左侧选择元素，并在右侧输入以下信息：

Introduction:

Description——“This cheat sheet will show you how to add an item to the Favorites view.”

Item 1:

Title——“Select a file”

Description——“Select the file that should be a favorite.”

15.5.3 注册备忘单

你已经创建了备忘单文件，但还没有在插件清单中注册新的备忘单。点击备忘单编辑器右上角的Register this cheat sheet链接（图15-26）。当Register Cheat Sheet对话框显示时（图15-27），点击New...以创建一个新的“收藏夹”备忘单类别，然后在Register Cheat Sheet对话框中输入以下内容以注册备忘单：

Category——“Favorites”

Description——“This cheat sheet will show you how to add an item to the Favorites view.”

点击Finish将在插件清单中注册备忘单文件并在org.eclipse.ui.cheatsheets插件中添加一个依赖项。现在我们必须要在插件清单文件中做出一些额外的更改。在帮助项目中打开plugin.xml文件并切换到Extensions页。在所有可用扩展点列表中展开org.eclipse.ui.cheatsheets.cheatSheetContent（图15-28），并选择新的备忘单类别和新的备忘单以显示它们的属性，这样我们可以根据以下内容对它们进行修改。

Favorites Category:

id——“com.qualityeclipse.favorites.cheatSheets”

备忘单类别的id。

name——“Favorites”

备忘单类别的名称。

Favorites Cheat Sheet:

id——“com.qualityeclipse.favorites.cheatsheet”

备忘单的id。

name——“Adding a favorite to the Favorites view”

备忘单的名称。

category——“com.qualityeclipse.favorites.cheatSheets”

放置备忘单的类别。

contentFile——“cheatsheets/FavoritesCheatSheet.xml”

备忘单内容文件的路径。

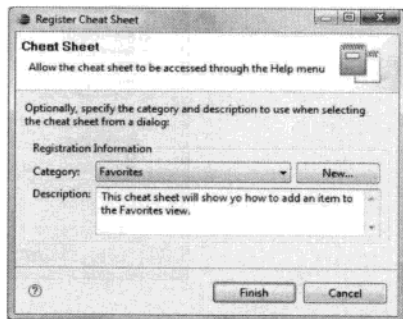


图15-27 注册备忘单对话框

PDF

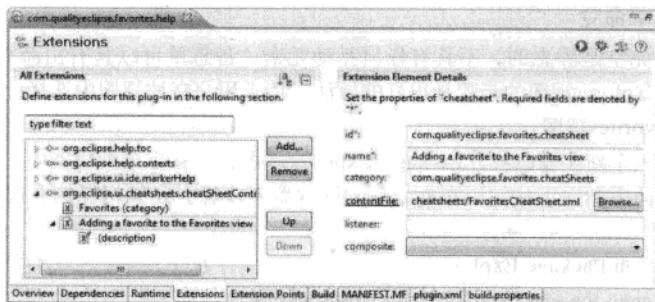


图15-28 显示收藏夹备忘单属性的扩展项元素细节

如果你切换至插件清单编辑器的plugin.xml页，你将看到定义新的备忘单和类别的XML的以下新片段：

```
<extension point="org.eclipse.ui.cheatsheets.cheatSheetContent">
  <category
    id="com.qualityeclipse.favorites.cheatSheets"
    name="Favorites" />
  <cheatsheet
    category="com.qualityeclipse.favorites.cheatSheets"
    composite="false"
    contentFile="cheatsheets/FavoritesCheatSheet.xml"
    id="com.qualityeclipse.favorites.cheatsheet"
    name="Adding a favorite to the Favorites view">
    <description>
      This cheat sheet will show you how to add an item
      to the Favorites view.
    </description>
  </cheatsheet>
</extension>
```

一旦你已经完成备忘单的定义，通过启动运行时工作台测试它。打开Cheat Sheet Selection对话框（图15-29），选择Favorites > Adding a favorite to the Favorites view项，然后点击OK按钮。

新建的收藏夹备忘单将出现于一个新视图中，并且它的介绍在一开始将是展开的（图15-30）。

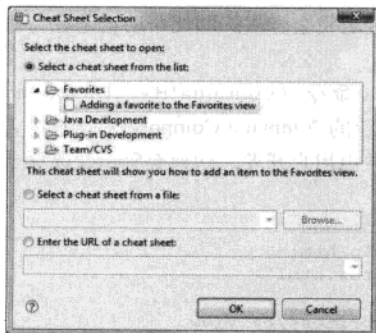


图15-29 显示收藏夹备忘单的备忘单选择对话框

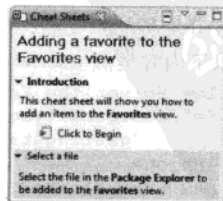


图15-30 收藏夹备忘单

15.5.4 添加备忘单命令

此刻，备忘单完全是静态的，不具有任何互动元素。备忘单可以具有由命令定义的活动内容。添加打开Package Explorer视图的步骤是很好的做法，请求用户在该视图中选择一个文件，并最终将选中文件添加至Favorites视图。

我们首先创建一个新步骤以打开Package Explorer视图。在备忘单编辑器中，点击Add Step按钮（图15-26）并使用向上和向下按钮以在介绍后立即放置新步骤。在编辑器的右边，为新步骤输入以下值。

- Title——“Open Package Explorer”
- Body——“Open the Package Explorer view.”

我们通过关联Show View命令打开Package Explorer视图作为该步骤的一部分。在Command字段的右边，点击在这些字段下面的Browse...按钮。当Command Composer对话框出现后，在左边的树中选择Views > Show View，然后在右边的View参数字段中选择Package Explorer并点击OK。一旦完成，如果你切换至备忘单编辑器中的源代码选项卡，新的步骤将与以下内容类似（serialization属性必须全部位于一行内，并且不能具有空格）。

```
<item title="Open Package Explorer">
  <description>Open the <b>Package Explorer</b> view.</description>
  <command
    required="false"
    serialization=
      "org.eclipse.ui.views.showView(
        org.eclipse.ui.views.showView.viewId
        =org.eclipse.jdt.ui.PackageExplorer)" />
</item>
```

通过点击Add Step按钮添加另一个步骤，并使用向上和向下按钮以在末尾放置新的步骤。在编辑器的右边，为新步骤输入以下值。

- Title——“Select the Favorites > Add command.”
- Body——“Add the selected file to the Favorites view.”
- Help Type——Help Document Link
- Help Value——“/com.qualityeclipse.favorites.help/html/gettingstarted/adding_favorites.html”

Help Type和Help Value字段是可选的，并用于使用备忘单任务链接帮助页。该链接必须对于包含帮助页的插件的名称，然后是插件相对于帮助页的路径是完全合格的即使帮助页是在与备忘单同样的插件，链接也必须是完全合格的。

主插件已经具有一个我们可以在这里重用的Add命令（com.qualityeclipse.favorites.commands.add，参见6.1.1节），但它并没有出现于上面讨论的Command Composer对话框中，这是因为主插件没有在开发环境中安装，而只是在工作区打开并由用户开发。切换至Source选项卡并修改最后一项，与以下内容类似（serialization全部在一行）。

```
<item title="Select the Favorites > Add command."
  href="/com.qualityeclipse.favorites.help/html
    /gettingstarted/adding_favorites.html">
  <description>
    Add the selected file to the <b>Favorites</b> view
  </description>
  <command
```

```

        required="false"
        serialization=
            "com.qualityeclipse.favorites.commands.add
            (com.qualityeclipse.favorites.command.sourceView
            =org.eclipse.jdt.ui.PackageExplorer)"/>
    </item>

```

15.5.5 添加命令参数

当用户点击最后一个步骤的Click to perform链接时，Cheat Sheet视图将获得焦点，以使没有可以被添加至Favorites视图的项被选中。为了修复这一问题，上面的serialization属性指定Add命令应通过传递一个新的sourceView参数使用来源于Package Explorer视图的选择。我们必须修改Add命令以添加一个可选的commandParameter。这可以通过右键点击com.qualityeclipse.favorites.commands.add命令（参见6.1.1节）并选择实现New > commandParameter。修改命令以使它与以下内容类似。

```

<command
  categoryId="com.qualityeclipse.favorites.commands.category"
  description="Add selected items to the Favorites view"
  id="com.qualityeclipse.favorites.commands.add"
  name="Add">
  <commandParameter
    id="com.qualityeclipse.favorites.command.sourceView"
    name="Source View"
    optional="true"
    values="org.eclipse.ui.internal.registry.ViewParameterValues">
  </commandParameter>
</command>

```

一旦参数已经被添加至命令，那么我们必须修改AddToFavoritesHandler处理器（参见6.3.1节）以理解该新参数并将焦点移至在执行Add命令之前的指定视图。

```

public Object execute(ExecutionEvent event)
    throws ExecutionException {

    String viewId =event.getParameter(
        "com.qualityeclipse.favorites.command.sourceView");
    if (viewId !=null){
        IWorkbenchWindow window
            =HandlerUtil.getActiveWorkbenchWindow(event);
        try {
            window.getActivePage().showView(viewId);
        }
        catch (PartInitException e){
            FavoritesLog.logError(e);
        }
    }

    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection)
        FavoritesManager.getManager().addFavorites(
            ((IStructuredSelection) selection).toArray());
    return null;
}

```

然后，你需要添加一个新项至备忘单定义，因此，添加以下内容至FavoritesCheatSheet.xml文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<cheatsheet title="Adding a favorite to the Favorites view">
  ...
  <item
    href="/com.qualityeclipse.favorites.help
      /html/gettingstarted/adding_favorites.html"
    title="Select the Favorites >Add command">
    <action
      pluginId="com.qualityeclipse.favorites"
      class="com.qualityeclipse.favorites.actions.
        AddToFavoritesAction"/>
    <description>
      Select the Favorites >Add command.
    </description>
  </item>
  ...
</cheatsheet>
```

一旦你已经添加多个步骤至备忘单，那么你可以如稍早那样测试它。在完成打开视图和选择文件的头两个步骤之后，将出现第三个步骤（图15-31）。使用Click to Perform 按钮以执行该任务并添加选中文件至Favorites视图。点击Open Related Help 按钮将打开与添加项至Favorites视图相关的帮助页。

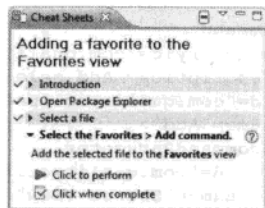


图15-31 显示第二个任务的收藏夹备忘单

15.6 RFRS相关事项

《RFRS Requirements》的“帮助”一节包含了9个（1个要求和8个最佳做法）与帮助相关的内容。

15.6.1 通过帮助系统提供帮助（RFRS 3.7.2）

要求#1说明：

扩展项的Eclipse用户界面帮助应通过与它集成的Eclipse帮助系统提供。这包括用于启动外部集成厂商工具的接口（菜单项、工具栏条目等）。你工具的与Eclipse集成不那么紧密的组件的帮助可以通过任意它可能使用的已有的帮助系统提供。

为了达到该要求，打开你的帮助插件的插件清单并指出org.eclipse.help.toc扩展点的使用。对于Favorites视图来说，指向plugin.xml文件的以下行。

```
<extension point="org.eclipse.help.toc">
  <toc file="toc.xml" primary="true"/>
  <toc file="tocgettingstarted.xml"/>
  <toc file="tocreference.xml"/>
</extension>
```

然后，打开Eclipse Help窗口并展示你的插件的帮助文档出现于主要主题列表。对于Favorites视图而言，你应展示收藏夹指南出现在列表中（图15-13）。

如果你的插件包含了不是通过Eclipse帮助系统提供的在线帮助，也在这里展示它。

15.6.2 通过帮助系统提供所有帮助（RFRS 5.3.7.1）

最佳做法#2说明：

通过集成至Eclipse UI的Eclipse帮助系统提供你插件的所有帮助。

如同要求 #1 中那样, 打开你的帮助插件的插件清单并指出org.eclipse.help.toc扩展点的使用。打开Eclipse Help窗口并展示你的帮助文档是可用的。该最佳做法实际上是对要求#1的进一步提炼。唯一的区别是, 为了通过该测试, 你的插件应通过Eclipse帮助系统提供它所有的在线帮助。

15.6.3 使用F1激活上下文帮助 (RFRS 5.3.7.2)

最佳做法#3说明:

如果上下文帮助可用, 应通过F1激活。对于紧密集成的产品, 这项内容要求使用一个或多个用于构建用户界面的SWT或JFace窗口小部件关联帮助。

为了通过这项测试, 提供对按下F1你的插件将显示上下文相关的infopop窗口的描述。对于Favorites视图而言, 你将展示与该视图关联的infopop窗口 (图15-15)。

15.6.4 实现活动帮助 (RFRS 5.3.7.3)

最佳做法#4说明:

为主题实现活动帮助。这些主题通过使用工作台操作最好地进行阐述。比如, 考虑一个名为“导入外部插件”的主题。不告诉用户使用工作台并选择File > Import, 然后选择External Plug-ins and Fragments并点击Next, 主题应简单地说“点击这里以打开导入外部片段向导”。链接应调用一个你已经定义的类。该类稍后应在该页打开向导。

通过这项测试比它要求实现一个或多个活动帮助元素要难得多。展示你的活动帮助元素如何回调你的插件以启动不同的向导或其他命令。为了了解更多关于创建活动帮助的信息, 参见在线Eclipse文档中包含的《Platform Plug-in Developer Guide》的“活动帮助”主题。

15.6.5 独立帮助的使用 (RFRS 5.3.7.4)

最佳做法#5说明:

如果帮助不是紧密地集成于Eclipse UI, 那么使用独立的帮助或一个基于网络服务器的信息中心。

这项测试大体上与最佳做法#2相对, 因此它们两者只能通过一个测试 (好消息是它们不都是被列为要求)。对于这项测试, 说明由你的程序提供的任意不基于工作台的帮助。对于Favorites视图而言, 稍早添加的网页访问按钮可能是合格的。

15.6.6 附加文档的使用 (RFRS 5.3.7.5)

最佳做法#6说明:

如果提供了附加文档 (除了readme文件之外), 它应被包含于用于实现产品或提供集成帮助的插件目录的其中一个。这可能是在一个\doc子目录或一个具有名称可能为的co.tool.doc插件目录。

对于最后添加文档或其他产品指南, 如果没有将其添加至集成文档, 那么应被包含于插件目录的readme文件中。

为了通过这项测试, 展示所有由你的插件提供的附加文档, 如readme文件或评估指南。

15.6.7 提供任务流的概述 (RFRS 5.3.5.34)

最佳做法#7说明:

给出一个任务流的概述。一句或两句描述当用户完成备忘单中的所有步骤时, 将“产生/生成/创建”什么。除此之外, 关于简单场景或代码的信息可以在这里添加。将信息分为单独的段落。

为了通过这项测试, 访问你插件定义的一个备忘单并展示它包含了一个介绍性段落。该段落提

供了一个任务流的概述。

15.6.8 仅说明一个任务 (RFRS 5.3.5.35)

最佳做法#8说明:

备忘单的每一个步骤必须仅说明一个任务并包含使用仅一个工具 (向导、对话框或编辑器等)。

a. 如果任务使用超过一个工具, 你应实现每一个步骤仅使用一个工具。

b. 如果步骤要求用户不启动任意工具执行操作, 那么它是一个手动步骤, 项描述必须指导用户在完成任任务时按下合适的按钮。

为了通过这项测试, 访问由你的插件定义的一个备忘单并展示每一个步骤仅说明了一个任务并且仅使用了一个工具。

15.6.9 为每一个步骤提供帮助链接 (RFRS 5.3.5.36)

最佳做法#9说明:

每一个步骤必须具有一个包含关于步骤中描述的任务的信息的帮助链接。该帮助链接显示附加信息以帮助用户理解任务, 完成该步骤, 或获得关于该任务的更高级选项。更多的帮助链接可以放置于启动页。

为了通过该测试, 访问由你的插件定义的一个备忘单并展示它包含了一个具有每一步骤不同的信息的帮助链接。

15.7 总结

在介绍了Eclipse帮助系统之后, 这一章说明了如何创建并集成你自己的在线帮助。它展示了如何使你的帮助可以在Eclipse Help窗口中可用和使用上下文相关的F1键。本章还讲述了如何使用备忘单在复杂任务中指导你的用户。

参考文献

本章资源(2.9节).

Adams, Greg, and Dorian Birsan, "Help Part 1, Contributing a Little Help," August 9, 2002 (www.eclipse.org/articles/Article-Online%20Help%20for%202_0/help1.htm)

Ford, Neal, "Centralizing Help in Eclipse," ThoughtWorks, June 21, 2005 (www-128.ibm.com/developerworks/opensource/library/os-eclipsehelp)

Zink, Lori, "Understanding Eclipse Online Help," HP, February 2005 (devresource.hp.com/drc/resources/eclipsedoc/index.jsp)

"Dynamic Content in Eclipse Help System Pages," Eclipse.org, December 2005 (www.eclipse.org/eclipse/platform-ua/proposals/xhtml/HelpDynamicContent.html)

Eclipse Help: **Platform Plug-in Developer Guide > Programmer's Guide > Plugging in help**

第16章 国际化

如果插件开发者的用户不仅在一个国家，如美国，那么国际化将成为开发的一个重要方面。Eclipse和底层的Java运行时环境都提供了用于分隔语言、UI相关事项与代码的API。这一章包含相关技术并提供了国际化示例插件的例子。

每一个程序（Eclipse插件也不例外）包含数十个可读字符串。这些字符串在窗口、对话框和菜单中向用户展示信息。隔离这些字符串以使它们可以被本地化（翻译）为不同国家的语言。这是国际化插件的最重要的步骤。

向插件用户展示他们自身的字符串来源于不同类型的文件。插件清单文件包含视图和透视图的名称，以及菜单和操作的标签。插件的about.ini文件（在第18章中有更多细节）包含在Eclipse About对话框中显示的文本。

其他在插件接口中可见的字符串，如小部件标签和错误消息文本，来源于实现插件的Java类。有不同技术和工具以用于外部化这些位于不同文件的字符串。

i18n 有时候，术语国际化被缩写为“i18n”，这是因为在第一个“i”和最后一个“n”之间有18个字母。

16.1 外部化插件清单

插件清单文件包含不同字符串以用于标识插件的元素。一些字符串，如插件标识符和与扩展项关联的唯一ID，不需要被翻译，因为它们不会向用户显示。实际上，翻译标识符和唯一ID将有可能破坏你的插件。其他字符串，如视图名称和操作的标签，需要被翻译，因为用户将看到它们。

外部化来源于插件清单文件的可读字符串是简单的。文件plugin.properties（一个你将创建的标准Java资源包文件）包含被释放的字符串。作为示例，我们首先以Favorites插件清单的以下片段开始讲解。

```
<plugin
...
<extension point="org.eclipse.ui.views">
    <category
        name="Quality Eclipse"
        id="com.qualityeclipse.favorites">
    </category>
    <view
        name="Favorites"
        icon="icons/sample.gif"
        category="com.qualityeclipse.favorites"
        class="com.qualityeclipse.favorites.views.FavoritesView"
        id="com.qualityeclipse.favorites.views.FavoritesView">
    </view>
</extension>
...
</plugin>
```


粗体显示的行包含需要被释放的字符串。其他行包含不需要被释放的文本，如类名、标识符、文件名或版本号。

每一个字符串被一个首先一个百分号(%)描述键所替代。这些是将被用于相关的plugin.properties文件的同样的键。唯一的规则是需要键在插件内是唯一的。你还应尝试给予这些键以描述性名称，以使它们在plugin.xml和plugin.properties文件中能很容易地被认出。

在释放之后，片段将与以下内容类似：

```
<plugin
...
  <extension point="org.eclipse.ui.views">
    <category
      name="%favorites.view.category.name"
      id="com.qualityeclipse.favorites">
    </category>
    <view
      name="%favorites.view.name"
      icon="icons/sample.gif"
      category="com.qualityeclipse.favorites"
      class="com.qualityeclipse.favorites.views.FavoritesView"
      id="com.qualityeclipse.favorites.views.FavoritesView">
    </view>
  </extension>
...
</plugin>
```

plugin.properties文件将与以下内容类似：

```
# Contains translated strings for the Favorites plug-in
favorites.view.category.name=Quality Eclipse
favorites.view.name=Favorites
```

我们从以下内容开始释放过程：选择两个收藏夹插件项目，右键点击并在上下文菜单中选择PDE Tools > Externalize Strings...命令。当Externalize Strings对话框出现时，在左侧选择com.qualityeclipse.favorites > plugin.xml，然后编辑右侧的键以给予它们描述性名称。

当字符串已经被释放至plugin.properties文件时，就可以翻译它们了。每一种目标语言的翻译文件应被命名为plugin_<language>_<country>.properties。在这里，<language>和<country>代表用于表示语言和国家（国家是可选的）的双字母代码（ISO 639和ISO 3166）。

提示 ISO 639语言代码列表可以在以下地址找到：www.unicode.org/onlinedat/languages.html
ISO 3166国家代码可以在以下地址找到：www.unicode.org/onlinedat/countries.html

比如，标准德语翻译应被命名为plugin_de.properties，并且与以下内容类似：

```
# Enthält übersetzten Text für die steckbaren Lieblingen
favorites.view.category.name= Qualitätseklipse
favorites.view.name=Lieblinge
```

同样地，标准法语翻译应被命名为plugin_fr.properties，并且与以下内容类似：

```
# Contient le texte traduit pour les favoris plugin
favorites.view.category.name= Éclipse De Qualité
favorites.view.name=Favoris
```

16.2 外部化插件字符串

当插件清单已经被外部化后，另一个可读字符串的主要来源是插件的Java源代码。在收藏夹示例内部，有数十个向用户展示的字符串。这些字符串以UI元素和消息的形式出现。

为了展示外部化Java源代码文件中的字符串的过程，以下内容将带你浏览从FavoritesView类释放字符串的全过程。Favorites视图包含几个硬编码的字符串。这些字符串被用于UI元素，如菜单标签和表列头（图16-1）。

在FavoritesView类内部，你应聚焦于以下硬编码字符串定义（使用Refactor > Extract Constant...命令释放至常量）：

```
private static final String TYPE_COLUMN_LABEL = "";
private static final String NAME_COLUMN_LABEL = "Name";
private static final String LOCATION_COLUMN_LABEL = "Location";
private static final String OPEN_FILTER_COMMAND_LABEL = "Filter...";
```

Eclipse包含一个强大的字符串外部化工具。该工具将完成大部分工作。我们首先选择FavoritesView类。然后，选择Source > Externalize Strings...命令以打开Externalize Strings向导（图16-2）。

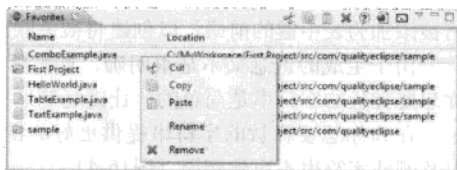


图16-1 显示不同的字符串的收藏夹视图



图16-2 外部化字符串向导

该向导扫描类以查找所有字符串文本并在Strings to externalize列表中展示它们。表的第一列用于决定字符串是被翻译、从不翻译还是跳过（直到下一次在该类上运行向导）。键列包含向导的第一次为字符串生成唯一键的尝试（从1至n）。值列包含发现的字符串。在表中选择一个条目将在下面的文本格的上下文环境中高亮显示它。

在向导的顶部, 点击Use Eclipse's string externalization mechanism选项以使用新的字符串外部化机制。common prefix字段已经被使用字符串所在的类的名称进行了预填充。将该值更改为“FavoritesView_”(请注意下划线而不是句号的使用, 以使每一个键都是合法的Java标识符)。该值将被添加为表中键的前缀, 以创建将被关联至每一个字符串的最终键。

由于生成的键意义不是很明确, 首先要做的事是编辑它们以表示它们将要替代的字符串。由于你想要替代的字符串是简单的, 让键复制这些值(图16-3)。

在为你想要释放的字符串提供更好的键之后, 你需要检查列表并标识出哪些字符串将被释放, 以及哪些字符串不应被翻译(图16-4)。

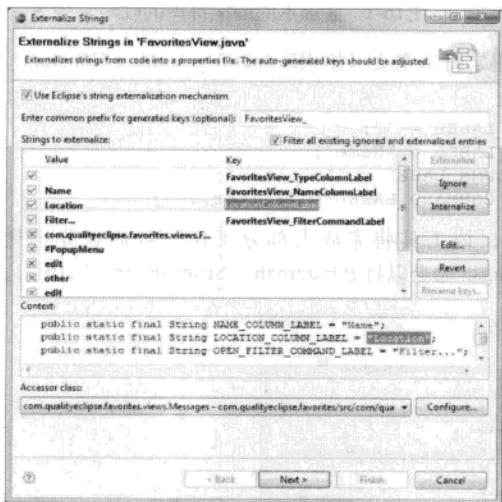


图16-3 使用更好的键名外部化字符串向导



图16-4 外部化字符串向导将字符串标记为“忽略”(Ignore)

默认地, 所有字符串都将被释放。这意味着每一个字符串和它的键一起, 将被添加至属性文件。在Java源文件内部, 字符串将被来源于属性文件的对应字符串生成的字段引用所替代。如果你知道一个字符串不应被释放, 那么在开发中使用一个特殊的行末尾注释(比如, //\$NON-NLS-1\$)手动标记它, 以使字符串释放在以后变得较为容易。

注释末尾的数字表示当一行有超过一个的字符串时, 哪一个字符串应被忽略。如果某一行超过一个的字符串应被忽略, 每一个字符串将在该释放格式(包括开头的双斜杠)中具有它自己的行末尾注释。

选择一个字符串并点击Internalize按钮将告诉向导使用上面使用的行末尾注释标记该字符串为不可翻译的。点击Ignore按钮将使得向导不采取任何操作。或者, 你可以点击表的第一列的单选框图像。图像将在三个选项间循环: ☒ Externalize, ☒ Internalize和 ☐ Ignore。

提示 在你如何编写你的代码上请仔细考虑, 由于它可以生成超过严格必需的需要外部化的字符串。使用单个字符替代单个字符串, 寻找机会以重用键而不是创建新的键, 并使用消息绑定以降低需要被外部化的字符串的数目。比如, 假设“Count”已经被外部化, 你可能遇到以下三个场景:

```
// Bad, we don't want to externalize "Count ("
label.setText("Count (" + count + ")");
// Good, we already have "Count" externalized.
label.setText("Count" + " (" + count + ')'); //$NON-NLS-2$
// Better, use binding patterns whenever possible.
label.setText(
    MessageFormat.format("Count {1}",
        new String[] {count})
```

在第二个场景中，你可以重用分配至“Count”的键并减少所需的键的数量。在第三个场景，你可以创建一个新键。该键将一个动态参数编码至一个相对于翻译的位置（在其他语言，%1参数可能出现该字符串的其他地方）。

点击Configure...按钮将打开一个对话框。在该对话框中你可以指定字符串将被释放的位置，以及如何访问它们（图16-5）。Package字段指定属性文件将被创建的位置，Property file name字段指定将创建的属性文件的名称。它将默认为messages.properties。除非你有理由做其他工作，否则你应该简单地接受默认值。

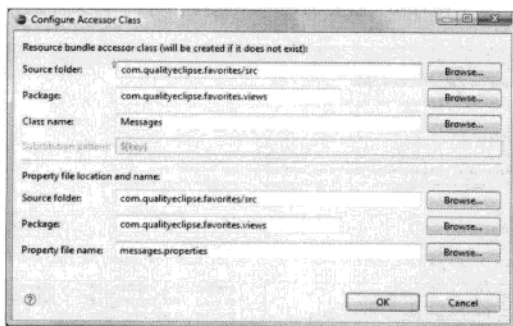


图16-5 定义资源包和访问设置

默认地，资源包访问器类将使用名称Message创建。该类将定义对应于每一个释放的字符串的字段，以及填入这些字段必需的代码。

如果你在使用较老的Eclipse字符串释放机制，并且不需要创建该类，那么将Class name字段设为空。如果你这么做（可能因为你需要使用一种不同的机制），你还可能需要指定一个不同的string substitution pattern（默认样式被设计为匹配该向导将创建的访问器类）。

由于Externalize Strings向导实际上是使用Eclipse重构框架创建的，之后的两个页面对于所有重构向导来说都是一样的。第一页（图16-6）将展示所有错误或信息类消息。在这里你将看到的唯一消息是一个通知。它说明属性文件不存在并且需要被创建。点击Next按钮以继续。

向导的最后一页将展示所有将要实施的建议更改的列表（图16-7）。首先，你将看到FavoritesView类的所有将生成的字符串替补。在那之后，你将看到属性文件和资源包访问器类的内容。

点击Finish按钮将实现所有建议的更改。你之前关注的FavoritesView的原始行将被以下内容替代：

```
private static final String TYPE_COLUMN_LABEL
    = Messages.FavoritesView_TypeColumnLabel;
private static final String NAME_COLUMN_LABEL
```

```

        = Messages.FavoritesView_NameColumnLabel;
private static final String LOCATION_COLUMN_LABEL
        = Messages.FavoritesView_LocationColumnLabel;
private static final String OPEN_FILTER_COMMAND_LABEL
        = Messages.FavoritesView_OpenFilterCommandLabel;

```

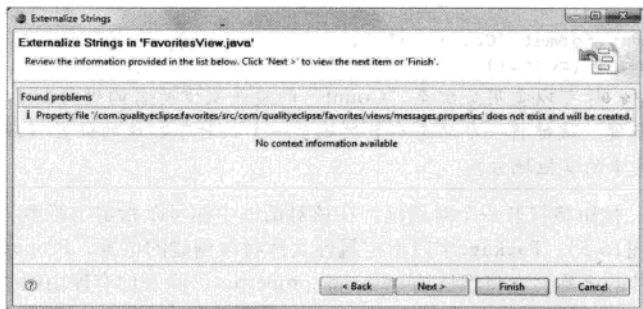


图16-6 需要创建messages.properties文件

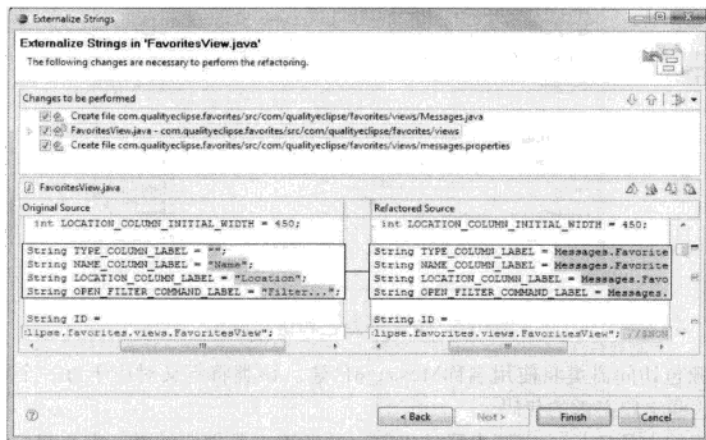


图16-7 查看建议更改

每一个字符串文本将被一个至Messages类中定义的匹配字段的引用所替代。每一个字段将被使用它的对应字符串的值填充。任意在一开始被忽略的字符串文本将被使用//\$NON-NLS-1\$标签标记。该标签将阻止它们在以后被释放。

资源包访问器类的代码将与以下内容类似：

```
package com.qualityeclipse.favorites.views;
```

```
import org.eclipse.osgi.util.NLS;
```

```
public class Messages extends NLS {
    private static final String BUNDLE_NAME =
        "com.qualityeclipse.favorites.views.messages"; //$NON-NLS-1$

```

```
public static String FavoritesView_LocationColumnLabel;  
public static String FavoritesView_NameColumnLabel;  
public static String FavoritesView_OpenFilterCommandLabel;  
public static String FavoritesView_TypeColumnLabel;  
static {  
    // initialize resource bundle  
    NLS.initializeMessages(BUNDLE_NAME, Messages.class);  
}  
private Messages() {  
}  
}
```

最终, messages.properties文件将与以下内容类似:

```
FavoritesView_LocationColumnLabel=Location  
FavoritesView_NameColumnLabel=Name  
FavoritesView_OpenFilterCommandLabel=Filter...  
FavoritesView_TypeColumnLabel=
```

与plugin.properties文件中一致, 每一种目标语言的翻译后的属性文件应被命名为<basename>_<language>_<country>.properties。在这里, <language>和<country>表示用于表明语言和国家的双字母代码, 而<basename>是原始属性文件的名称。

提示 为了确保你已经外部化你插件的所有字符串 (或标记它们为不可翻译的), 考虑将Non-externalized strings选项从“Ignore”更改为“Warning”。这些选项位于Java > Compiler > Errors/Warnings首选项页。或者, CodePro包含一个字符串文本代码审计规则。该规则将标识硬编码的字符串文本 (使用选项以忽略单字符串、仅包含空格或数字的字符串、static final字段初始程序、匹配某种样式的字符串等)。

当外部化字符串时, 以下建议已被证明是有用的:

- 移除任意标点符号, 如键中出现的“&”。如果你需要在你的键内分隔单词, 使用特定字符以标准化, 如句号、破折号或下划线。
- 编辑.properties文件并保持条目是分好类的。首先基于条目适用的文件对条目进行分类, 然后在文件内根据字母顺序对它们进行进一步分类。
- 输出所有通用值并创建一个通用键。移动所有通用键至文件中它们自己所在部分。为你的通用键添加common_前缀并将它们移至文件的顶部。这将减少翻译的数量并消除了翻译中发生变更的可能性。
- 当你编辑键时, 努力将它们与原语言字符串保持尽可能紧密, 因为这将使Java代码和XML对于原语言开发者更容易阅读。如果你决定这么做, 尽量在原语言字符串被更改时重命名键, 否则这将导致混淆。当然, 数字类型的键不存在这种问题。
- 当原字符串包含冒号时, 如“Name:”, 生成的键将包含两个下划线。不要定义与此类似的键, 相反地, 回到原字符串并更改“Name:”为“Name”+“:”。这不仅将键保持简单, 并且它还保证冒号不会在翻译时丢失。这里唯一的问题是你是否真正需要遵守本地标点符号规则。然而, 这可能会是相当棘手的。
- 你应总是考虑包含一个具有国际化错误消息的错误数字, 以使在一种语言中显示的错误能在其他语言中表达。
- 在生成的静态getString(String key)方法 (由较老的Ecilpse字符串释放机制使用) 中, 编辑

catch语句以包含以下行。这些行将在终端窗口记录所有丢失的资源键。这比在你的程序中寻找“!<key>!”要简单得多。

```
System.err.println(e.getMessage());
```

16.3 使用片段

主插件可以包含所有翻译后的属性文件，但这在译文将在产品本身之后发布的情形下还不够理想。如果所有的翻译后的文件需要包含于主插件中，你要么需要延后主插件的发布，要么当译文可用时重新发布它。幸运的是，Eclipse包含一个名为插件片段（plug-in fragment）的机制。该机制提供了该问题的一个很好的解决办法。

片段被用于扩展另一插件的功能。片段一般用于提供不同的语言包、维护更新和平台特定的实现类。当载入片段时，它的特性和功能被移至那些基础插件，以使它们出现为来源于基础插件自身。

开发者不需要知道插件或它的一个片段是否添加特定资源，这是因为创建类载入器透明地处理该事项。这使得发布插件文件的独立于主插件的翻译版本（比如，HTML、XML、INI和属性文件）是比较容易的。

属性文件在资源包命名规则的后面（稍早已说明）。其他资源，如HTML、XML和INI文件被放置于nl/<language>或nl/<language>/<country>目录。在这里，<language>和<country>表示用于表明语言和国家的双字母代码（该目录结构在15.2.3节中首次介绍）。

提示 片段还可以用于缓和不同Eclipse版本间的区别，与用于访问内部类和方法（参见21.2.6节）。

16.3.1 新建片段项目向导

使用新建片段项目向导创建一个新的片段是比较容易的。从File菜单，选择New > Project以启动新建项目向导（图16-8）。在该向导的第一页，选择Plug-in Development > Fragment Project，然后点击Next按钮。

在向导的下一页（图16-9），输入项目的名称。在这里，应当是“com.qualityeclipse.favorites.nl1”，与插件片段标识符一致。Eclipse惯例是使用基础插件的名称加上后缀“.nl1”命名一个添加国家语言支持至基础插件的插件片段项目。点击Next按钮。

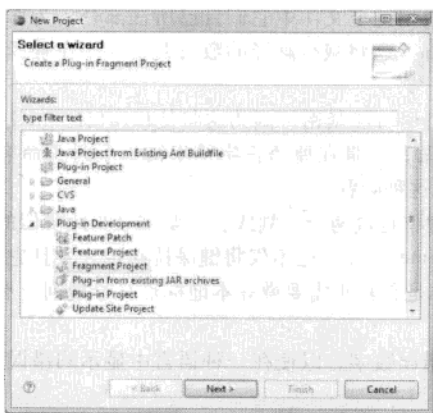


图16-8 新建项目向导中选择片段项目

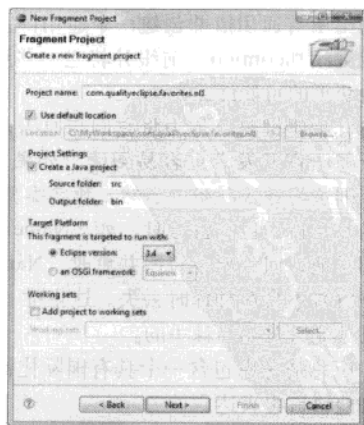


图16-9 新建片段项目向导

Fragment Content页(图16-10)提供了字段以命名片段, 设置它的ID和版本号, 并标识插件ID和它扩展的版本。在需要时使用Browse...按钮以选择插件。在这里, 你需要扩展已有的com.qualityqualityeclipse.favorites插件(图16-11)。

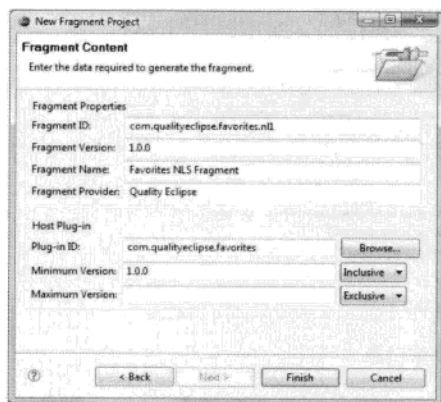


图16-10 初始片段文件的所需数据

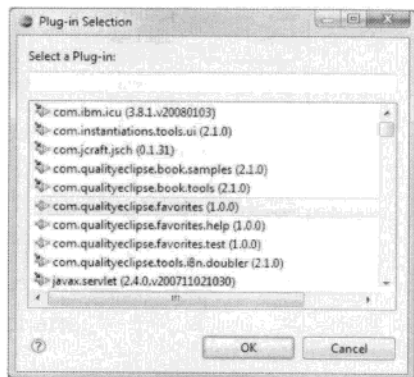


图16-11 插件选择向导

在Plug-in ID字段的下面是Minimum Version和Maximum Version字段。这些字段可以用于指定片段与基础插件的哪一个版本兼容。通过选择合适的最小和最大版本, 以及不同的包含与排除组合, 可以达到以下兼容性级别:

- 完全兼容 (Perfect) 意味着基础插件精确地匹配提供的版本号。
- 等价兼容 (Equivalent) 意味着版本可能在服务或限定符级别不同。
- 兼容 (Compatible) 意味着插件可能具有一个更新的较小版本号。
- 更大或相等 (Greater or Equal) 意味着插件可能具有任意更新的版本号。这是应被指定的一般的兼容性级别。

输入“1.0.0”至Minimum Version字段, 选择Inclusive选项并点击Finish按钮以完成向导并生成片段清单文件。

16.3.2 片段清单文件

双击片段清单文件中的MANIFEST.MF, 将打开片段清单编辑器(图16-12)。编辑器与插件清单编辑器十分类似, 因为它也包含了概览 (Overview)、依赖项 (Dependencies)、运行时 (Runtime)、扩展点 (Extension Points) 等页面。

如果你切换至Runtime页面(图16-13), 你将看到在片段的运行时路径中没有包括任何内容。不同的文件, 如翻译后的HTML、XML和INI文件, 基于相关的位置(语言 and 国家的组合)位于特殊命名的子目录。

为了让这些目录出现于运行时classpath, 点击New...按钮以添加一个“New Library”条目至列表。右键点击该条目并将它重命名为“\$nls/”。这是将使系统在运行时基于当前位置替代正确的classpath条目的特殊语法。如果位置被设置为德国, 作为示例, “\$nls”应被“de”替代, 并且“de/”子目录将被添加至运行时路径。

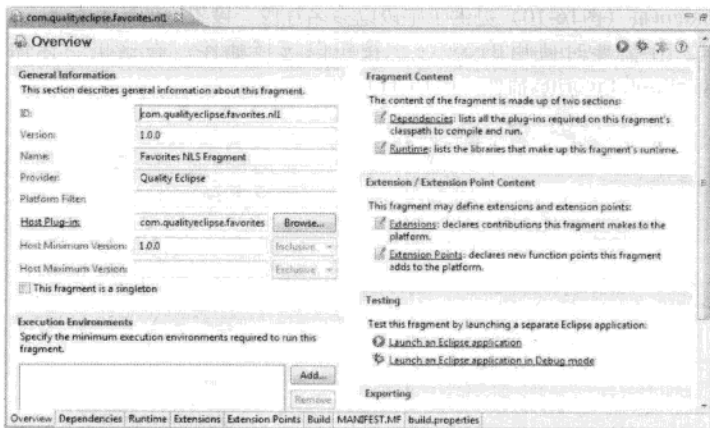


图16-12 片段清单编辑器

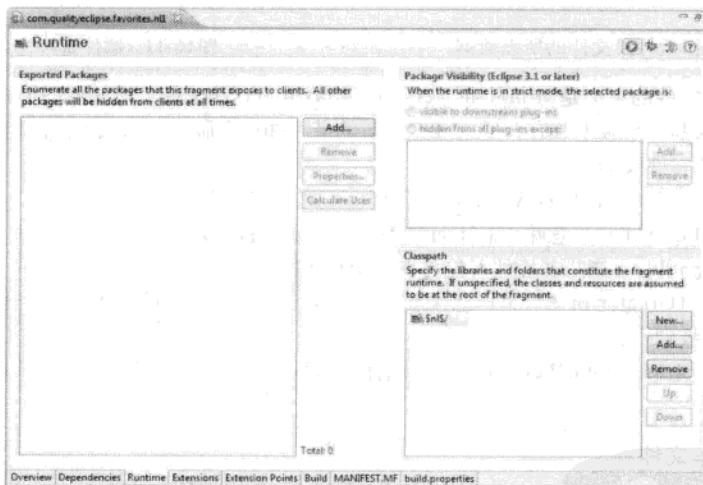


图16-13 片段运行时信息页

然后，让我们切换至MANIFEST.MF页并看一看生成的文本：

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Favorites NLS Fragment
Bundle-SymbolicName: com.qualityeclipse.favorites.nls
Bundle-Version: 1.0.0
Bundle-Vendor: QualityEclipse
Fragment-Host: com.qualityeclipse.favorites;
    bundle-version="1.0.0"
Bundle-ClassPath: $nls/
```

该内容与你在插件中期望看到的十分类似。第一个主要的区别是片段不具有它们自身的插件生命周期（它们遵循它们相关联的插件的生命周期），因此，它们不需要它们自己的Bundle-Activator属性。第二个区别是片段不声明任意它自己的依赖项（它们继承于相关插件）。

在完整插件中的这些相关的有趣属性是Fragment-Host和bundle-version属性。Fragment-Host标识了该片段将扩展的插件。bundle-version指定了该片段希望可以扩展的目标插件版本。

重复该过程以创建第二个片段项目，并命名为“com.qualqualityeclipse.favorites.help.nl1”以包含Favorites帮助文件翻译。

16.3.3 片段项目内容

你需要做的最后一件事是添加所有翻译后的文件至合适的片段项目文件夹中的合适的目录（图16-14）。在我们这里，不同的翻译后的toc*.xml和*.html帮助文件应被放置于com.qualqualityeclipse.favorites.help.nl1片段项目并且所有的其他的翻译后的文件应被放置于com.qualqualityeclipse.favorites.nl1项目。

假设你需要提供不同文件的德语和法语翻译，项目将具有nl/de和nl/fr目录以容纳所有的翻译后的HTML、XML和INI文件（与15.2.3节中提到的类似）。

plugin.properties文件的翻译版本被放置于片段的根目录，而messages.properties文件的翻译版本将被放置于com.qualqualityeclipse.favorites.views目录。

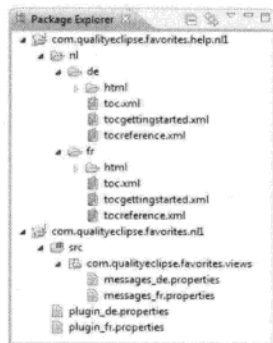


图16-14 片段项目的内容

16.4 手动测试

为了测试你程序的国际化，使用合适的国家语言参数启动程序。比如，为了测试标准德语翻译，打开Run Configurations或Debug对话框（参见2.6.1节），切换至Arguments选项卡并修改项目参数为包含“-nl de”。为了测试标准法语翻译，修改项目参数为包含“-nl fr”。

16.5 总结

为了让插件可以被世界用户所使用，它应被国际化。释放插件的可读字符串至一种可以容易被翻译的格式是最重要的步骤。如同本章中描述的那样，Eclipse提供了一些工具以实现该功能。Externalize Strings向导使得从Java代码释放字符串变得容易，而片段提供了一种便利的打包机制以独立于主插件传递翻译后的内容。

参考文献

本书资源 (2.9节)。

Java internationalization tutorial (java.sun.com/docs/books/tutorial/i18n/intro/index.html).

Internationalization (I18n) (java.sun.com/j2se/1.4.2/docs/guide/intl/).

Kehn, Dan, Scott Fairbrother, and Cam-Thu Le, “How to Internationalize Your Eclipse Plug-in,” IBM, August 23, 2002 ([eclipse.org/articles/Article Internationalization/how2I18n.html](http://eclipse.org/articles/Article%20Internationalization/how2I18n.html)).

Kehn, Dan, "How to Test Your Internationalized Eclipse Plug-in," IBM, August 23, 2002 (eclipse.org/articles/Article-TVT/how2TestI18n.html).

Kehn, Dan, Scott Fairbrother and Cam-Thu Le, "Internationalizing Your Eclipse Plug-in," IBM, June 1, 2002 (www-128.ibm.com/developerworks/opensource/library/os-i18n).

ISO 639 language codes (www.unicode.org/onlinedat/languages.html).

ISO 3166 country codes (www.unicode.org/onlinedat/countries.html).

Eclipse Help: **Java Development User Guide > Tasks > Externalizing Strings**



第17章 创建新扩展点

Eclipse通过定义扩展点（extension point）降低了改进的难度，但该技术不只适用于Eclipse自身。每一个插件可以定义它自己的扩展点。这些扩展点可以在内部被作为一种规范和灵活的编程方法，或者在外部作为第三方插件以一种受控的、松耦合的、灵活的方式改进一个已有插件的方法。这一章讨论相关的API，并提供了关于创建扩展点的示例。这些扩展点可以使第三方扩展插件的功能。

17.1 扩展点机制

到此刻为止，我们已经作为一个用户讨论了扩展点；现在，我们需要深入研究幕后机制，使我们最终可以创建自己的扩展点让其他人使用。扩展点不仅让产品更灵活，并且通过仔细查看插件的特定方面，你可以使产品更灵活和可定制。我们的目标是让用户使用你的产品并做一些从未展望过的事。

扩展点在Eclipse中作为一个松耦合的功能块广泛使用。插件在它的插件清单中声明扩展点，显示接口和相关类的最小集以让其他人使用。其他插件声明该扩展点的扩展项，实现合适的接口，并且引用提供的类或基于提供的类进行创建（图17-1）。

每个扩展点都具有一个由插件唯一标识符、一个英文句点和一个仅包含数字字母和下划线的简单标识符组成的唯一标识符。当声明扩展点（参见17.2.1节）时，仅适用简单标识符。当声明扩展点的扩展项（参见17.5节）时，使用该扩展点的完整标识符。

每个扩展点可以具有一个定义它们应如何被使用的模式（schema）。虽然该模式对于恰当的扩展点使用不是必需的，但是Eclipse PDE可以为扩展项执行基本自动验证，以及为扩展点自动生成类Javadoc文档时使用该模式。

模式是一个XML格式的文件，传统上具有名称<extension-point-id>.exsd，并位于插件的安装目录的一个模式子目录中。比如，本章稍后讨论的扩展点模式位于<Eclipse_install_dir>/plugins/com.qualityeclipse.favorites_1.0.0/schema/favorites.exsd。

17.2 定义扩展点

在Favorites产品中，你可能需要让其他插件扩展产品以提供Favorites对象的附加类型。为了完成该目标，创建一个新的favorites扩展点和模式，以及其他类可以扩展的相关基础结构类型。作为该过程的一部分，重铸（recast）当前Favorites对象为该新扩展点的扩展项以向你证明自己证明新扩展点确实可以工作。

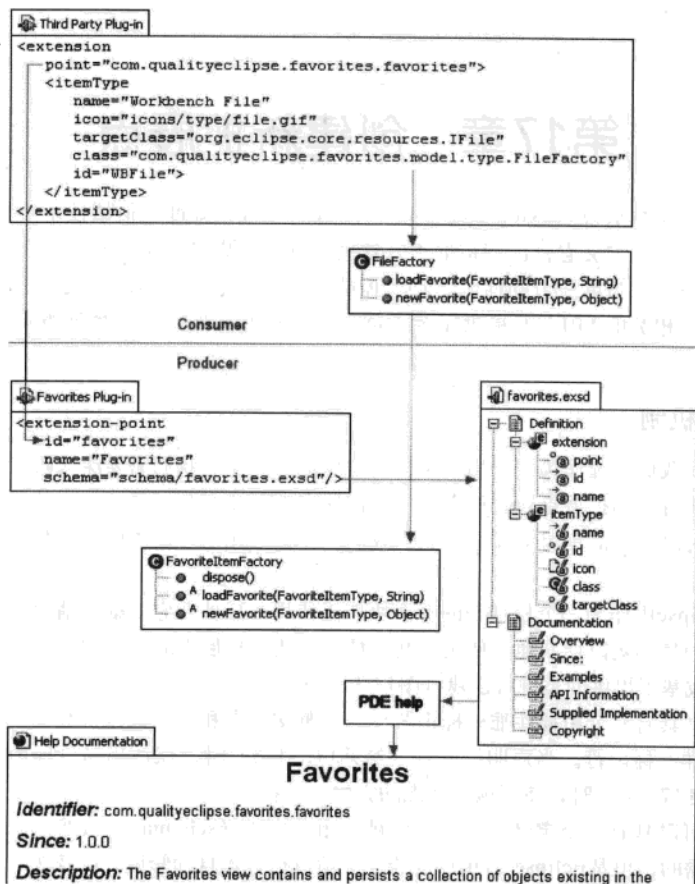


图17-1 扩展点概述

17.2.1 创建扩展点

我们首先打开Favorites插件清单编辑器，并切换至扩展点页。点击Add...按钮以打开New Extension Point向导，然后输入“favorites”作为标识符，“Favorites”作为名称（图17-2）。

点击Finish以创建新扩展点并打开模式文件（更多关于模式的内容位于17.2.2节）。当切换回插件清单编辑器之后，扩展点页应展示刚定义的扩展点（图17-3）。

切换至插件清单编辑器的plugin.xml页将显示一个指定标识符的新扩展点声明、可读名称和模式的相对路径。

```

<extension-point
  id="favorites"
  name="Favorites"
  schema="schema/favorites.exsd"/>

```

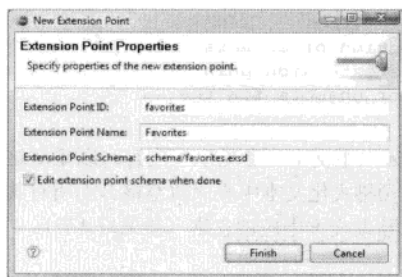


图17-2 新建扩展点向导

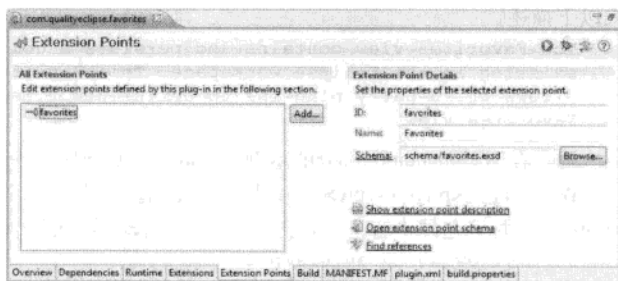


图17-3 插件清单编辑器中的扩展点页

上述声明指定了扩展点的本地标识符。完整标识符是插件标识符加上本地标识符。它在引用扩展点时使用。在这里，完整标识符是`com.qualityeclipse.favorites.favorites`。

17.2.2 创建扩展点模式

New Extension Point向导自动打开模式编辑器以编辑位于Favorites项目的模式目录（图17-4）新创建的`favorites.exsd`文件。如果你需要再一次打开模式编辑器，你可以浏览至模式目录并双击`favorites.exsd`文件，或在Favorites插件清单中选择`favorites`扩展点，然后选择Open Schema。

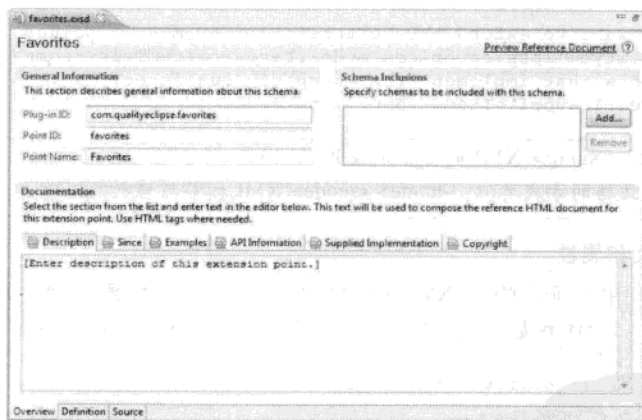


图17-4 Eclipse扩展点模式编辑器

模式编辑器有几个主要部分：概览（Overview）页的主要信息（General Information）和文档（Documentation），以及定义（Definition）页的扩展点元素（Extension Point Elements）和扩展项元素细节（Extension Element Details）。主要信息部分包括了插件ID（Plug-in ID）、点ID（Point ID）和点名称（Point Name）。

Extension Point Elements列表包含元素及其相关的属性。这些属性作为扩展点的扩展项出现（参见17.2.3节）。

作为开始，列出目标，以使你对打算完成的任务有一个清晰的认识。扩展点模式被PDE用于为扩展点动态生成帮助页（参见17.4节）。在模式编辑器的描述（Description）部分（在概览页），输

入以下描述。

The Favorites view contains and persists a collection of objects existing in the Eclipse workspace. This Favorites extension point allows third-party plug-ins to define new types of objects for the Favorites view.

通过在Documentation部分的示例(Examples)选项卡并输入以下文本(参见以下内容)以重复这一过程。请注意

```
和
```

 XML标签的使用。这些标签指出预格式化文本中应保留空格,而所有其他的在这些标签之外,超过一个空格的空白在自动生成的HTML帮助文本中应被忽略(参见17.4节)。

The following is an example
of the Favorites extension point usage:

```
<p>
<pre>
    <extension point="com.qualityeclipse.favorites.favorites">
        <itemType
            id="com.example.xyz.myNewFavoriteItemId"
            name="New Favorites Item Name"
            class="com.example.xyz.MyFavoriteItem"
            targetClass="com.example.xyz.MyObjectClass"/>
        </extension>
    </pre>
</p>
```

然后,为API Information输入以下内容:

Plug-ins that want to extend this extension point must subclass
<samp>com.qualityeclipse.favorites.model.FavoriteItemType</samp> and
generate objects that implement the <samp>com.qualityeclipse.-
favorites.model.IFavoriteItem</samp> interface.

请注意 <samp> </samp> XML的标签的使用,以表示一些语句中的代码。这些语句与许多Eclipse插件生成文档的方式类似。<code> </code> XML标签应当也能工作。

17.2.3 扩展点元素和属性

扩展点元素对应于扩展项声明中的XML元素。扩展点属性对应于扩展项声明中的XML属性。比如,在下列扩展项声明中,itemType是一个扩展点元素,而id、name、class和targetClass是扩展点属性。

```
<extension point=
    "com.qualityeclipse.favorites.favorites">
    <itemType
        id="com.example.xyz.myNewFavoriteItemId"
        name="New Favorites Item Name"
        class="com.example.xyz.MyFavoriteItem"
        targetClass="com.example.xyz.MyObjectClass"/>
    </extension>
```

扩展点属性具有几个不同的与它们相关的属性。在模式编辑器中选择一个属性将在模式编辑器的Attribute Details部分显示它的属性(图17-5)。该扩展项属性的属性是:

- Name——属性的名称,出现于扩展项声明中。比如,在上面的扩展项声明中,id、name、class和targetClass都是属性名。
- Deprecated——表示属性是否为弃用的。
- Use——表示该属性在扩展项中是需要的(required)从而必须被明确声明,还是可选的

(optional), 表示它可以从扩展项声明中省去。或者, Use可以被声明为默认的 (default), 表示如果它没有被明确声明, 那么它将默认为稍后列出的Value属性指定的值。

- Type——属性的类型: boolean、string、java或resource。在这里, 这些是由模式编辑器和PDE识别的仅有的四种类型。如果属性仅应为true或false, 那么选择boolean; 如果属性应为Java类, 那么选择java; 如果属性表示一个文件, 那么选择resource; 对于所有其他属性, 选择string。
- Extends——如果上述的Type属性是java, 那么该属性表示必须扩展的类的完整合格名称。
- Implements——如果Type是java, 那么该属性表示必须实现的接口的完整合格名称。
- Translatable——如果Type是string, 该布尔值表示该属性是否是可读的并且是否应被翻译。
- Restrictions——如果Type是string, 那么该属性可以限制属性值为一个枚举 (enumeration) 或合法字符串的离散列表。比如, 使用该字段, 你可以指定一个属性仅可以为值 “one”、“two” 或 “three”。
- Description——属性的文档。
- Value——如果Use被指定为default, 那么该属性表示如果该属性没有在扩展项声明中明确指定时将被使用的默认值。

对于Favorites产品而言, 你需要该扩展点的扩展项以指定关于正在被添加的Favorites项的类型的信息。以下是需要包含的扩展点属性。

- name——用户可读名称。
- id——唯一标识符。
- icon——图像的相对路径 (可选的)。
- class——用于创建该类型的项的FavoriteItemFactory。
- targetClass——被该类型包装的对象的类型。

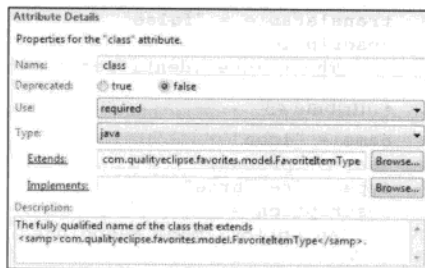


图17-5 显示属性的属性值的属性细节区域

提示 如果你的扩展点打算用于通用目的, 那么请考虑惰性扩展项初始化, 以使你不会载入不必要的插件, 以避免内存膨胀和让系统变慢。比如, 在这里, targetClass被用于确定Favorites项类型是否应在定义它实际被载入的插件之前被使用。如果你在扩展项没有指定该信息, 那么当用户在Favorites视图上拖放新对象时, 它们将需要载入并查询每一个收藏夹类型以确定哪一种类型应处理放下的对象。这可能潜在地载入超过必需的插件, 使工作区膨胀并变慢。作为替代, 你根据targetClass对类型进行预扫描以确定哪一种类型可能可以处理被放入的对象。如果你的扩展点仅打算用于你自己的插件, 那么你不需要额外的复杂性和与代理相关的系统开销。

首先, 你需要把一个代表正被定义的Favorites项类型的itemType元素添加到添加这些属性的对象。为了完成该任务, 切换至Definition页, 点击New Element按钮, 选择被创建的新_element1, 并通过在Element Details区域编辑它的Name属性更改名称为 “itemType”。

当在模式编辑器中选择itemType时, 点击New Attribute按钮以添加一个新属性至itemType元素。将它的名称改为 “name”。当模式编辑器中选择name属性时, 输入 “A human-readable name for this type of Favorites object.” 作为名称属性的描述。

重复该过程以添加四个其他属性。当你完成时, 你应为具有这里指定的属性的itemType元素已

经定义了以下属性。

- attribute #1

```
name = "name"
use = "optional"
type = "string"
translatable = "true"
description =
    "A human-readable name for this type of Favorites object."
```
- attribute #2

```
name = "id"
use = "required"
type = "string"
translatable = "false"
description =
    "The unique identifier for this type of Favorites object."
```
- attribute #3

```
name = "icon"
use = "optional"
type = "resource"
description =
    "An option image associated with this type of Favorites object."
```
- attribute #4

```
name = "class"
use = "required"
type = "java"
extends =
    "com.qualityeclipse.favorites.model.FavoriteItemFactory"
description =
    "The fully qualified name of the class that extends
    <samp>com.qualityeclipse.favorites.model.
    FavoriteItemFactory</samp>."
```
- attribute #5

```
name = "targetClass"
use = "required"
type = "string"
translatable = "false"
description =
    "The fully qualified name of the class wrapped by this item
    type. This is not the class name for the IFavoriteItem object
    returned by either
    <samp>FavoriteItemType.loadFavorite(String)</samp> or
    <samp>FavoriteItemType.newFavorite(Object)</samp>, but
    rather the object wrapped by that IFavoriteItem object that
    causes the IFavoriteItem.isFavoriteFor(Object) to return true."
```

提示 扩展项应如何为你的扩展点提供行为？你是否需要扩展项以实现接口或扩展一个抽象基础类？如果你需要扩展项以实现接口，那么你应该承诺给予扩展项作者更多的灵活性。不足之处是，任意对于该接口的更改将破坏已有的扩展项。作为替代，如果你需要扩展项以扩展一个抽象基础类，那么你应该在保留松耦合的优点的同时，保持一些灵活性。向抽象基础类添加

一个实体方法将不会破坏已有的扩展项。假设给你机会以更改以后实现中的API，请不要牺牲太多扩展项作者的灵活性。如果你确认你的API将不会更改，那么接口是很好的实现方法。否则，一个抽象基础类将给予你改进API所需要的灵活性。

如果你需要抽象基础类无法给你的，而接口具有的灵活性那么请考虑使用接口但提供一个抽象基础类。该抽象基础类实现接口以让扩展项在选择时可以创建。一旦有了该方法，你可以通过在你的实现任意新接口方法的抽象基础类中添加实体方法更改接口API以及降低中断。所有使用抽象基础类的扩展项将不受你的接口API更改的影响，然而所有直接实现接口的扩展项必须修改以适应新的API。

17.2.4 扩展点元素语法

当已经定义了扩展点元素后，请创建元素语法。这些语法描述元素被组合的方式可由PDE验证。当你在左侧的Extension Point Elements列表选择一个扩展项元素时，所有相关的语法元素将在它下面嵌套显示。在左侧选择语法元素将在右侧显示它的细节属性。

不展示所有可能的语法元素，表17-1展示几个常见的场景和相关的语法。想要得到的XML结构出现于左侧。在右侧是用于描述具有方括号——[]包含的属性值的结构。

表17-1 XML语法

XML	语 法
<pre> <parentElement ... > <childElement ... /> </parentElement> </pre>	<pre> + Sequence childElement </pre>
<pre> <parentElement ... > <childElement ... /> <childElement ... /> ... </parentElement> </pre>	<pre> + Sequence childElement [minOccurs = "0"] [maxOccurs = "unbounded"] </pre>
<pre> <parentElement ... > <childElement1 ... /> <childElement1 ... /> ... <childElement2 ... /> <childElement2 ... /> ... </parentElement> </pre>	<pre> + Sequence childElement1 [minOccurs = "0"] [maxOccurs = "unbounded"] childElement2 [minOccurs = "0"] [maxOccurs = "unbounded"] </pre>
<pre> <parentElement ... > <childElement1 ... /> <childElement2 ... /> </parentElement> -或- <parentElement ... > <childElement1 ... /> <childElement3 ... /> </parentElement> </pre>	<pre> + Sequence childElement1 + Choice childElement2 childElement3 </pre>

无论何时任何人扩展该扩展点，你都需要一个或多个itemType元素作为扩展项声明的一部分。在Extension Point Elements模式编辑器中，在列表中选择并展开扩展点元素extension将显示Sequence元素。选择Sequence，然后右键点击并选择New > itemType。展开Sequence以使itemType作为一个后代显示于Sequence的下面，然后点击itemType以在Element Reference Details区域显示它的属性。紧挨着Max Occurrences，选中unbounded属性。当这些完成后，你应在Extension Point Elements列表中看到itemType (1-*) (图17-6)。

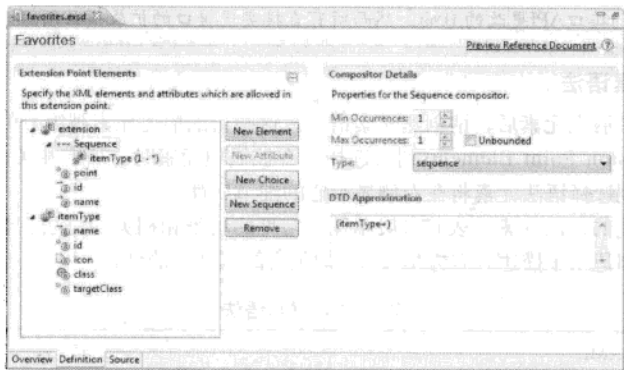


图17-6 显示收藏夹扩展点的模式编辑器

17.3 扩展点的后台代码

在已经定义了扩展点之后，你必须编写它的后台代码，以基于在扩展点的扩展项中声明的信息创建Favorites项类型和Favorites对象。根据Eclipse惰性初始化的主题，你需要将内存占用保持较低的水平，所以每一个包含它的Favorites项类型和插件必须仅当需要时才载入。为了达到这一目标，重构FavoriteItemType（参见17.2.3节）的一部分为一个新的FavoriteItemFactory，然后重新组织FavoriteItemType以根据扩展项信息创建类型。在这之后，是重铸Favorites项类型常量为新的Favorites扩展点的扩展项。

17.3.1 分析扩展项信息

对FavoriteItemType做出的第一个更改包括根据扩展项信息创建该类的实例，而不是在类中硬编码信息为常量。重命名TYPES数组为cachedTypes以更准确地代表该静态字段的目的。修改getTypes()方法为每一个发现的扩展项创建FavoriteItemType的一个新的实例。

```
private static final String TAG_ITEMTYPE = "itemType";

private static FavoriteItemType[] cachedTypes;

public static FavoriteItemType[] getTypes() {
    if (cachedTypes != null)
        return cachedTypes;
    IExtension[] extensions = Platform.getExtensionRegistry()
        .getExtensionPoint(FavoritesActivator.PLUGIN_ID, "favorites")
        .getExtensions();
```

```
List<FavoriteItemType> found
    = new ArrayList<FavoriteItemType>(20);
found.add(UNKNOWN);
for (int i = 0; i < extensions.length; i++) {
    IConfigurationElement[] configElements =
        extensions[i].getConfigurationElements();
    for (int j = 0; j < configElements.length; j++) {
        FavoriteItemType proxy =
            parseType(configElements[j], found.size());
        if (proxy != null)
            found.add(proxy);
    }
}
cachedTypes =
    (FavoriteItemType[]) found.toArray(
        new FavoriteItemType[found.size()]);
return cachedTypes;
}

private static FavoriteItemType parseType(
    IConfigurationElement configElement, int ordinal
) {
    if (!configElement.getName().equals(TAG_ITEMTYPE))
        return null;
    try {
        return new FavoriteItemType(configElement, ordinal);
    }
    catch (Exception e) {
        String name = configElement.getAttribute(ATT_NAME);
        if (name == null)
            name = "[missing name attribute]";
        String msg =
            "Failed to load itemType named "
            + name
            + " in "
            + configElement.getDeclaringExtension()
            .getNamespaceIdentifier();
        FavoritesLog.logError(msg, e);
        return null;
    }
}
```

提示 与以前一样，合适的异常处理是必需的，尤其是当通过扩展点处理松耦合代码时。在这种情况下，实例创建被包装于一个异常处理器中，以使一个不合适的声明的扩展项将不会使该方法失败，但作为替代，将生成一个包含足够信息以跟踪并更正问题原因的日志条目。

17.3.2 创建代理

接下来，你需要修改FavoriteItemType构造函数以从扩展项释放基本信息，而不载入声明该扩展项的插件。该实例是作为包含于声明插件中的工厂的代理。如果一个需要的属性丢失了，那么将抛出一个IllegalArgumentException，并被稍早描述的parseType()方法的异常处理器中捕获。

```
private static final String ATT_ID = "id";
private static final String ATT_NAME = "name";
```

```

private static final String ATT_CLASS = "class";
private static final String ATT_TARGETCLASS = "targetClass";
private static final String ATT_ICON = "icon";

private final IConfigurationElement configElement;
private final int ordinal;
private final String id;
private final String name;
private final String targetClassName;
private FavoriteItemFactory factory;
private ImageDescriptor imageDescriptor;

public FavoriteItemType(
    IConfigurationElement configElem, int ordinal
) {
    this.configElement = configElem;
    this.ordinal = ordinal;
    id = getAttribute(configElem, ATT_ID, null);
    name = getAttribute(configElem, ATT_NAME, id);
    targetClassName =
        getAttribute(configElem, ATT_TARGETCLASS, null);

    // Make sure that class is defined,
    // but don't load it.
    getAttribute(configElem, ATT_CLASS, null);
}

private static String getAttribute(
    IConfigurationElement configElem,
    String name,
    String defaultValue
) {
    String value = configElem.getAttribute(name);
    if (value != null)
        return value;
    if (defaultValue != null)
        return defaultValue;
    throw new IllegalArgumentException(
        "Missing " + name + " attribute");
}
}

```

提示 你如何确定从扩展项立即载入什么信息？什么应通过惰性初始化被延迟至一个访问器方法？载入扩展项属性值的方法，如 `IConfigurationElement.getAttribute(String)`，执行速度很快，因为它们返回已缓存的信息。其他方法，如 `IConfigurationElement.createExecutableExtension(String)`，是很慢的。这是因为它们在声明插件还没有载入时载入声明插件至内存。我们的原则是预先缓存并验证属性值，为大部分扩展项信息提供立即验证和“快速失败”（fast fail），但通过惰性初始化延迟所有可能导致载入声明插件的内容。

潜在地，每个扩展项可能是不合法的，并且你可能以没有由 `getTypes()` 方法返回的 `FavoriteItemType` 的合法实例而告终。为了减少这一问题，硬编码一个名为 `UNKNOWN` 的 `FavoriteItemType` 并添加它作为由 `getTypes()` 返回的集合的第一个对象。

```

public static final FavoriteItemType UNKNOWN =
    new FavoriteItemType()

```

```
{
    public IFavoriteItem newFavorite(Object obj) {
        return null;
    }
    public IFavoriteItem loadFavorite(String info) {
        return null;
    }
};

private FavoriteItemType() {
    this.id = "Unknown";
    this.ordinal = 0;
    this.name = "Unknown";
    this.configElement = null;
    this.targetClassName = "";
}
```

现在，修改访问器以获取基于已缓存的扩展项信息的项类型的信息。icon属性假定具有一个相对于声明插件的路径，而图像描述符根据该路径创建。图像占用宝贵的本地资源并相对缓慢地载入，因此它们根据当需要时载入原则进行惰性初始化。已载入的图像将被缓存，以使它们可以重用并在插件关闭时被正确地释放（参见7.7节以了解关于ImageCache的信息）。

```
private static final ImageCache imageCache = new ImageCache();

public String getId() {
    return id;
}
public String getName() {
    return name;
}
public Image getImage() {
    return imageCache.getImage(getImageDescriptor());
}
public ImageDescriptor getImageDescriptor() {
    if (imageDescriptor != null)
        return imageDescriptor;
    String iconName = configElement.getAttribute(ATT_ICON);
    if (iconName == null)
        return null;
    IExtension extension =
        configElement.getDeclaringExtension();
    String extendingPluginId = extension.getNamespaceIdentifier();
    imageDescriptor =
        AbstractUIPlugin.imageDescriptorFromPlugin(
            extendingPluginId,
            iconName);
    return imageDescriptor;
}
```

17.3.3 创建可执行扩展项

loadFavorite(String)和newFavorite(Object)方法根据扩展项中指定的重定向至factory对象。由于实例化factory对象包括载入包含它的插件，该操作将被延迟至需要时才执行。targetClassName方法用于确定是否需要载入相关的factory。实例化factory对象的代码包装于一个异常处理器中，以使日志可以记录关于发生的错误和哪些插件和扩展项相关的细节信息。

```
public IFavoriteItem newFavorite(Object obj) {
    if (!isTarget(obj)) {
        return null;
    }
    FavoriteItemFactory factory = getFactory();
    if (factory == null) {
        return null;
    }
    return factory.newFavorite(this, obj);
}

private boolean isTarget(Object obj) {
    if (obj == null) {
        return false;
    }
    Class<?> clazz = obj.getClass();
    if (clazz.getName().equals(targetClassName)) {
        return true;
    }
    Class<?>[] interfaces = clazz.getInterfaces();
    for (int i = 0; i < interfaces.length; i++) {
        if (interfaces[i].getName().equals(targetClassName)) {
            return true;
        }
    }
    return false;
}

public IFavoriteItem loadFavorite(String info) {
    FavoriteItemFactory factory = getFactory();
    if (factory == null) {
        return null;
    }
    return factory.loadFavorite(this, info);
}

private FavoriteItemFactory getFactory() {
    if (factory != null) {
        return factory;
    }
    try {
        factory = (FavoriteItemFactory) configElement
            .createExecutableExtension(ATT_CLASS);
    } catch (Exception e) {
        FavoritesLog.logError(
            "Failed to instantiate factory: "
            + configElement.getAttribute(ATT_CLASS)
            + " in type: "
            + id
            + " in plugin: "
            + configElement.getDeclaringExtension()
            .getNamespaceIdentifier(), e);
    }
    return factory;
}
```



提示 无论何时实例化扩展项中指定的对象，请总是使用 `IConfigurationElement.createExecutable(String)` 方法。该方法自动处理从插件清单中的扩展项至另一个插件中的运行时库的代码的引用，以及扩展项中指定的其他形式的后实例化（post-instantiation）的初始化（参见21.5节）的引用。如果你使用 `Class.forName(String)`，那么将只可以实例化你插件已经识别的对象，因为 `Class.forName(String)` 使用插件的类载入器，并因此将只实例化插件的 classpath 中的对象（参见21.9节以了解更多关于类载入器的内容）。

`newFactory` 类型是一个抽象基础类。它必须由其他提供新 `Favorites` 对象类型的插件扩展。参见17.2.3节的“提示”，以了解关于接口与抽象基础类的讨论。`factory` 类型包括一个实体 `dispose` 方法，以使子类可以在需要时执行清理，但如果不需要清理时可以不实现该方法。

```
package com.qualityeclipse.favorites.model;

public abstract class FavoriteItemFactory
{
    public abstract IFavoriteItem newFavorite(
        FavoriteItemType type, Object obj);

    public abstract IFavoriteItem loadFavorite(
        FavoriteItemType type, String info);

    public void dispose() {
        // Nothing to do... subclasses may override.
    }
}
```

17.3.4 清理

当插件关闭时，你必须释放所有已缓存图像并给予每一个 `factory` 对象一个清理的机会。添加 `disposeTypes()` 和 `dispose()` 方法至 `FavoriteItemType`。修改 `FavoritesActivator stop()` 方法以调用该新 `disposeTypes()` 方法。

```
public static void disposeTypes() {
    if (cachedTypes == null) return;
    for (int i = 0; i < cachedTypes.length; i++)
        cachedTypes[i].dispose();
    imageCache.dispose();
    cachedTypes = null;
}

public void dispose() {
    if (factory == null) return;
    factory.dispose();
    factory = null;
}
```

17.4 扩展点文档

一旦已经声明了扩展点和相关模式（参见17.2节），PDE将在所有已知插件扩展点的列表中包含他们。此外，作为模式的一部分添加的文档片段（参见17.2.2节和17.2.3节）被PDE动态创建至请求时的扩展点帮助页面。浏览至插件清单编辑器的 `Extensions` 页并点击 `Add...` 按钮。New Extensions向导包括了新的 `favorites` 扩展点（图17-7）。

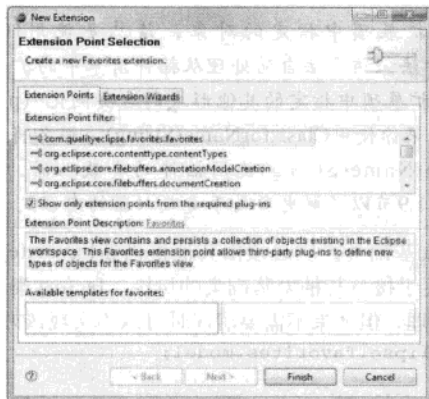


图17-7 显示收藏夹扩展点的新建扩展项向导

为了查看动态生成的帮助页，添加favorites扩展点至插件清单（参见17.5节），在插件清单中选择favorites扩展项，然后点击Show extension point description。这将打开一个浏览器以显示favorites扩展点的HTML帮助页（图17-8）。

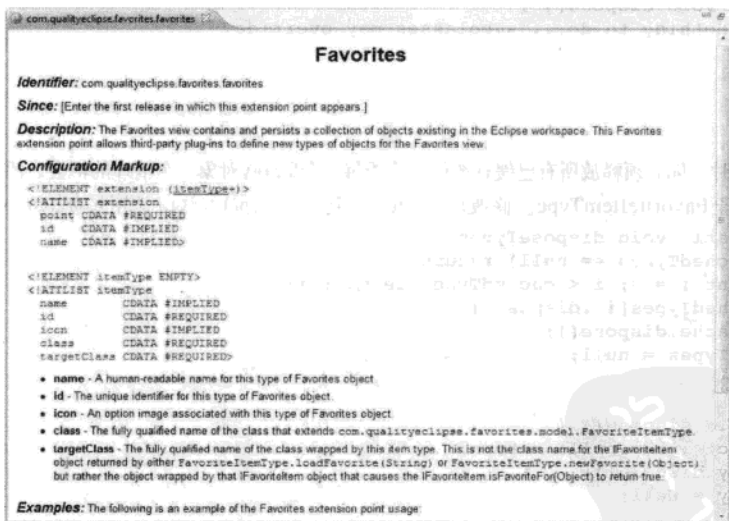


图17-8 收藏夹扩展点的动态生成的帮助

17.5 使用扩展点

我们已经重构了FavoriteItemType以使用来源于favorites扩展点的信息。所以，现在该类中的常量必须被重铸为扩展项并与factory类关联。该修改将帮助你测试你的新扩展点。

在插件清单编辑器的Extensions页，点击Add...按钮以打开New Extensions向导。当向导出现时，选择新的favorites扩展点（图17-7），然后点击Finish。

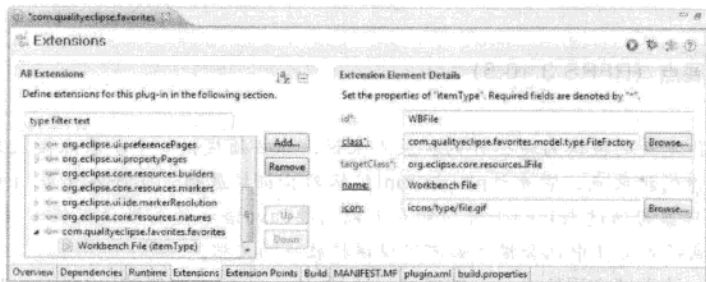


图17-9 显示工作台文件项类型属性的扩展项元素细节

一旦你已经创建了一个扩展项，你需要添加一个新的itemType来表示Workbench File项类型。右键点击新的收藏夹扩展项并选择New > itemType以添加一个新的Favorites项类型。点击新建的itemType并根据以下内容更改属性（图17-9）。

对于指定的icon属性，来源于Eclipse UI和JDT UI插件的图标已经被复制至Favorites插件。点击icon属性字段右侧的Browse...按钮将打开一个图像选择对话框，以便你可以从来源于插件中定义的图像为项类型选择合适的图像。

对于class属性，选择class属性字段左侧的class:标签以打开New Java Class向导。输入“com.qualityeclipse.favorites.model.type”作为包的名称，“FileFactory”作为类型名称。点击Finish按钮以生成该新类。从FavoriteItemType类中的WORKBENCH_FILE常量移除newFavorite()和loadFavorite()方法，以使新的FileFactory类与以下内容类似：

```
package com.qualityeclipse.favorites.model.type;
import ...
public class FileFactory extends FavoriteItemFactory
{
    public IFavoriteItem newFavorite(
        FavoriteItemType type, Object obj
    ) {
        if (!(obj instanceof IFile))
            return null;
        return new FavoriteResource(type, (IFile) obj);
    }
    public IFavoriteItem loadFavorite(
        FavoriteItemType type, String info
    ) {
        return FavoriteResource.loadFavorite(type, info);
    }
}
```

当完成时，这些Favorites项类型的第一个已经被从常量转换为扩展项。多次重复这一过程以重铸FavoriteItemType中的每一个常量Favorites项类型，除了稍早讨论的UNKNOWN项类型（参见17.3.2节）。

17.6 RFRS相关事项

《RFRS Requirements》的“扩展点”一节包含两个关于定义新扩展点的内容（一个要求和一個最佳做法）。

17.6.1 文档扩展点 (RFRS 3.10.5)

要求#1说明：

对于每一个你定义的被认为是公开的扩展点来说，你必须提供一个扩展点模式文件。通过使用扩展点模式描述你的扩展点，它允许plugin.xml编辑器验证扩展项，它还在扩展项的创建过程中提供帮助并运行编辑器为通过与Java平台功能交互以需要Java语法的属性设置值时提供帮助。此外，你还必须在扩展点模式文件中包含描述如何实现该扩展点的文档。

对于该测试，请尝试实现Favorites扩展点。通过plugin.xml编辑器的Extensions页添加扩展项并验证编辑器是否为应被添加至该扩展项的元素提供帮助。还需要核实你是否可以为Favorites扩展点打开一个提供关于如何正确地实现它的文档的html页面。

17.6.2 记录错误 (RFRS 5.3.10.1)

最佳做法#2说明：

注册表处理代码必须在日志中记录所有它在插件日志中监听到的错误。

展示注册表处理代码处理你插件扩展点的扩展项说明中的所有错误。对于Favorites扩展点而言，创建一个丢失name属性的扩展项。这将在Eclipse Error Log视图中创建一个条目（图17-10）。

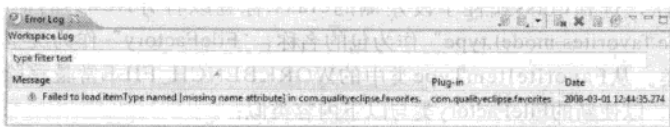


图17-10 显示扩展项说明中错误的错误日志

17.7 总结

扩展点是用于扩展Eclipse的主要机制。每一个Eclipse插件使用数十个扩展点以添加新的视图、操作、编辑器等。扩展点是不受限制的，不过，这仅对于Eclipse自身而言是成立的。你的插件既可以仅为内部使用而定义扩展点，也可以为其他第三方插件使用而定义。这一章从细节上阐述了创建和使用新的扩展点的过程。

参考文献

本书资源 (2.9节)。

D' Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer's Guide to Eclipse, Second Edition*, Addison-Wesley, Boston, 2004.

"Eclipse Platform Technical Overview," Object Technology International, Inc., February 2003 (www.eclipse.org/whitepapers/eclipse-overview.pdf).

Bolour, Azad, "Notes on the Eclipse Plug-in Architecture," Bolour Computing, July 3, 2003 (www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html).

第18章 功能部件、品牌化和更新

一个或多个Eclipse插件可以被分组至一个Eclipse功能部件 (feature), 以使用户可以容易地作为一个单元载入、管理和标记这些插件。这一章包含了有关Eclipse功能部件结构的概述, 并展示了如何使用内建的功能部件创建向导创建一个简单的功能部件。

这一章还讨论了使用功能部件以商业化或品牌化基于插件的产品，并在最后讨论了如何通过提供更新的网站打包并分发功能部件。

到目前为止，我们已经创建了几个插件。它们向Favorites视图添加了不同的功能部件。每一个插件与其他插件是松耦合的，并都没有展示任何统一结构或身份。功能部件提供了这种结构并为商标元素（如关于页和图像）提供了基础（图18-1）。

一旦被打包为功能部件，你将可以使用Eclipse更新管理器作为一个单元载入和卸载插件。

Eclipse 3.4新增内容 Eclipse 3.4包扩一个新的被称为“p2”的供给系统。该系统是较早版本的Eclipse中的更新管理器的全新版本。在大部分时候，你不需要担心p2的细节。你像以前一样继续创建插件、功能部件和更新站点，你可以使用一些附加步骤和p2一起优化你的更新站点（参见18.3节）。

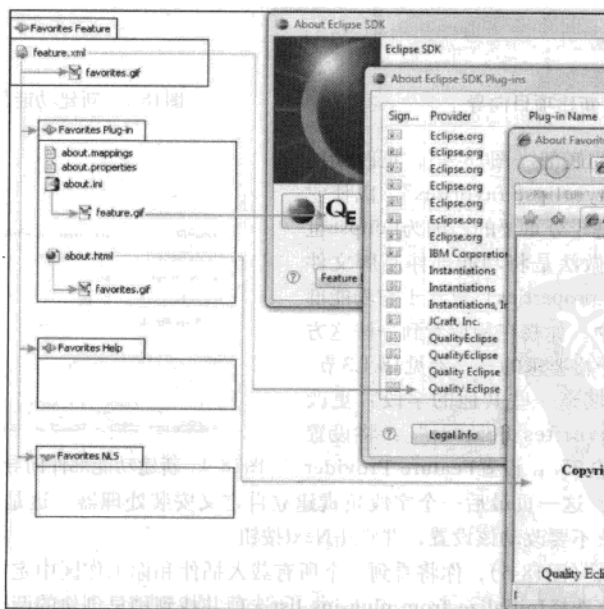


图18-1 功能部件文件关系与品牌化元素

18.1 功能部件项目

Favorites示例包括了四个项目（“Favorites插件”、“Favorites帮助”、“Favorites NLS片段”和“Favorites帮助NLS片段”）。你可能需要把这四个项目联合成一个功能部件。

18.1.1 创建新功能部件项目

为了创建新功能部件，你将首先使用New Project向导创建新的功能部件项目（Feature Project）（图18-2）。

在New Feature向导的第一页的顶部（图18-3），输入“com.qualityeclipse.favorites.feature”作为项目的名称。

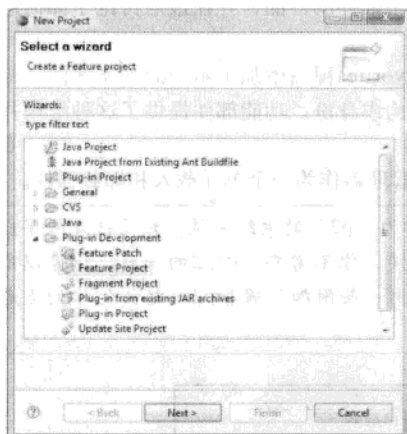


图18-2 新建项目向导

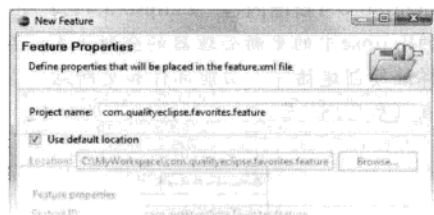


图18-3 新建功能部件向导

在向导的第一页的底部（图18-4），更改功能部件ID为“com.qualityeclipse.favorites”，以使它匹配主插件的ID。这是十分重要的。因为Eclipse世界中的被广泛接受的做法是将功能部件品牌文件（比如about.ini和about.properties）放置于与功能部件具有同样ID的插件中。你将在稍后看到一些这方面的内容。但这不是严格要求的——参见18.1.3节。

在这一页还需要填充一些其他的字段：更改Feature Name为“Favorites Feature”，将设置Feature Version为“1.0.0”，设置Feature Provider为“Quality Eclipse”。这一页最后一个字段负责建立自定义安装处理器。这是一个高级功能部件，将不会在本书中讨论。不要改动该设置，并点击Next按钮。

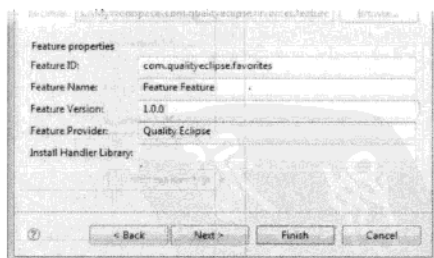


图18-4 新建功能部件向导的功能部件属性部分

在向导的最后一页（图18-5），你将看到一个所有载入插件和你工作区中定义的片段的列表，同时还有它们的版本号。选择Initialize from plug-ins list选项并找到稍早创建的两个插件和两个片段并选择它们。点击Finish以创建项目并生成它的功能部件清单文件。

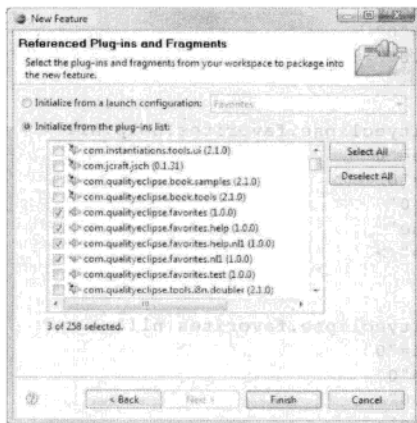


图18-5 新建功能部件向导引用的插件和片段页

请注意 如果你已经有一个指向你想要使用的插件的已有的启动配置，你可以通过使用 **Initialize from a launch configuration** 选项选择它。

18.1.2 功能部件清单文件

向导创建了一个有趣的文件：功能部件清单文件。基于向导中选择的选项，功能部件清单文件 (feature.xml) 将与以下内容类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
  id="com.qualityeclipse.favorites"
  label="Favorites Feature"
  version="1.0.0"
  provider-name="Quality Eclipse">
  <description url="http://www.example.com/description">
    [Enter Feature Description here.]
  </description>
  <copyright url="http://www.example.com/copyright">
    [Enter Copyright Description here.]
  </copyright>
  <license url="http://www.example.com/license">
    [Enter License Description here.]
  </license>
  <plugin
    id="com.qualityeclipse.favorites"
    download-size="0"
    install-size="0"
    version="0.0.0"
    unpack="false"/>
  <plugin
    id="com.qualityeclipse.favorites.help"
```



```
download-size="0"
install-size="0"
version="0.0.0"/>

<plugin
  id="com.qualityeclipse.favorites.help.n11"
  download-size="0"
  install-size="0"
  version="0.0.0"
  fragment="true"
  unpack="false"/>

<plugin
  id="com.qualityeclipse.favorites.n11"
  download-size="0"
  install-size="0"
  version="0.0.0"
  fragment="true"
  unpack="false"/>
</feature>
```

它的结构是比较简单的。在文件的开头，你将看到id、label、version和provider-name属性。description、copyright和license部分包含将要向用户展示功能部件的信息。

文件的剩余部分列出了组成该功能部件的独立插件和片段。每个插件由它的插件ID标识，而version属性指定作为该功能部件一部分的插件的指定版本。一般而言，被包含的插件的版本号应与功能部件的版本号相匹配。将fragment属性设置为true将标识所有被包含的片段。

18.1.3 功能部件清单编辑器

由向导生成的功能部件清单包含了定义功能部件需要的最基本的元素。可以定义很多其他属性以改进功能部件。功能部件清单编辑器提供了一个方便的接口以用于编辑功能部件的已有属性或添加新的属性。

双击功能部件清单文件feature.xml将打开功能部件清单编辑器（图18-6）。该编辑器与插件清单编辑器看起来十分相似，如Overview、Information、Plug-ins、Included Features、Dependencies、Installation、Build、feature.xml和build.properties页面。

在这一页需要做许多工作。一开始，ID、Version、Name和Provider字段将被基于至向导页的输入填充。在这里还有其他需要注意的字段。Branding Plug-in包括将包含功能部件品牌化文件的插件的名称。使用Browse...按钮以选择主插件或手动更改值为“com.qualityeclipse.favorites”，以使它与主插件的ID相匹配。

Update Site URL和Update Site Name字段用于指定将使用Eclipse更新管理器载入功能部件的更新站点的网址和名称。当更新管理器查找你插件的更新时，它将查看由你的更新URL定义的站点。请参考18.3节中的相关内容。

对于与公开Eclipse API相对的大部分插件来说，不同Eclipse平台的移植性将不是个问题。Eclipse不会阻止你使用平台特定的功能（比如Windows下的ActiveX支持）。在类似的情况中，你需要可以指定哪些环境对于你的插件是合适的。在Supported Environments部分，你可以为Operating Systems、Window Systems、Languages和Architecture提供一个逗号隔开的合法值的列表。

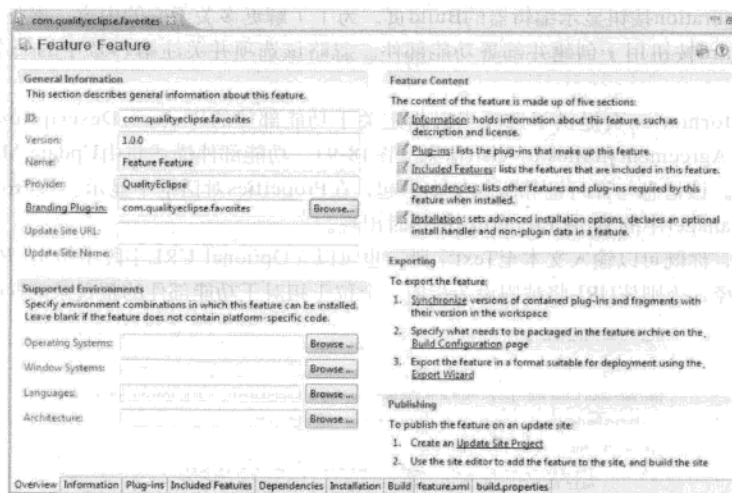


图18-6 功能部件清单编辑器

点击每一个字段右侧的Browse...按钮将打开一个适合于选中环境类型的选择对话框。比如，对于Operating Systems可用的选择包括aix、hpux、linux、macosx、qnx、solaris和win32（图18-7）。

在页面的右侧，Exporting部分包括了一些有趣的选项。Synchronize按钮用于同步被包含的插件和片段的版本号与功能部件的版本号。如果这些版本号不匹配，则更新管理器将无法正确地安装功能部件。点击按钮将打开版本同步（Version Synchronization）对话框（图18-8）。

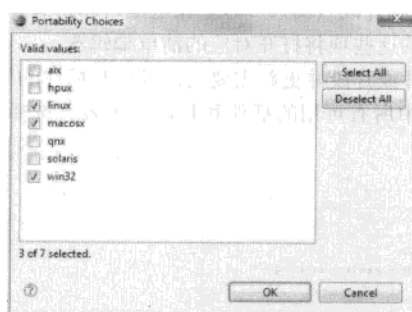


图18-7 用于操作系统的移植性选项

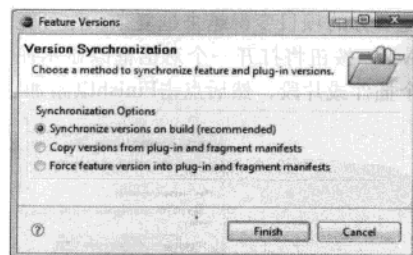


图18-8 版本同步对话框

该对话框包含三个选项。第一个，也是最有用的选项，是Synchronize versions on build (recommended)。这将更新所有已包括的插件和片段的清单文件，以使他们的版本号与功能部件的版本号匹配。第二个选项，Copy versions from plug-in and fragment manifests，将从每一个插件和片段复制单独的版本号并更新功能部件清单文件中对应的插件项。最后一个选项，Force feature version into plug-in and fragment manifests，完成相反的工作并获取功能部件清单文件中的每一个插件所定义的独立版本号并更新对应插件和片段的清单文件。选择第一个选项并点击Finish以返回至清单编辑器。

Build Configuration按钮显示编辑器的Build页。为了了解更多关于它的内容,参见19.2.10节。

Export Wizard按钮用于创建并部署功能部件。忽略该选项并关注第19章中的一个更复杂的构建操作。

编辑器的Information页提供了选项卡以指定关于功能部件的Feature Description, Copyright Notice, License Agreement和Sites to Visit信息(图18-9)。功能部件描述将由Update Manager在选中功能部件时显示。该信息与许可证和版权文本一起,在Properties对话框中显示。该Properties对话框在点击Update Manager中的Show Properties链接时出现。

对于这些项,你既可以输入文本至Text字段,也可以在Optional URL字段中指定URL。除非URL是站点的绝对路径,否则该URL将被假设为指向一个位于相对于功能部件的根目录的HTML文件。

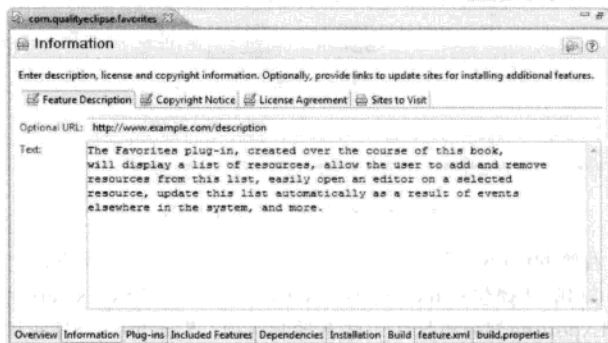


图18-9 描述、许可证和版权信息页

编辑器的Plug-ins页列出你的功能部件中包含的插件和片段(图18-10)。在这里,你将看到当第一次创建功能部件时选中的四个同样的选项。双击任意的这些项将打开对应的清单编辑器。

随着你的项目变得越来越复杂,你可能需要添加插件或片段并更新需要功能部件和插件的列表。点击Add...按钮将打开一个对话框以显示在你的工作区中所有可用的插件和片段的列表。选择一个或多个插件或片段,然后点击Finish以添加它们至列表。

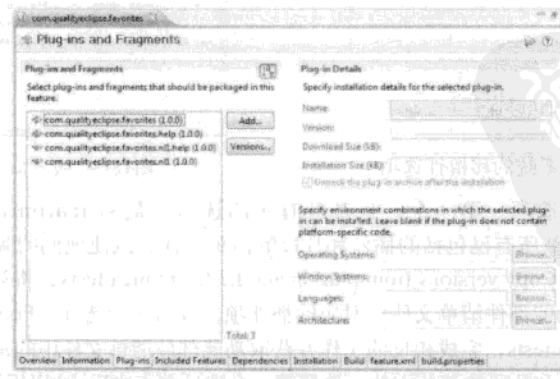


图18-10 插件和片段页

编辑器的Included Features页包含一个子功能部件的列表。这些子功能部件作为该功能部件的一部分（图18-11）。点击Add...按钮将允许你选择将成为当前功能部件的后代的功能部件。

如果一个功能部件的The feature is optional字段被选中，不需要在此时成功载入父功能部件（它可以在以后需要时载入和安装）。

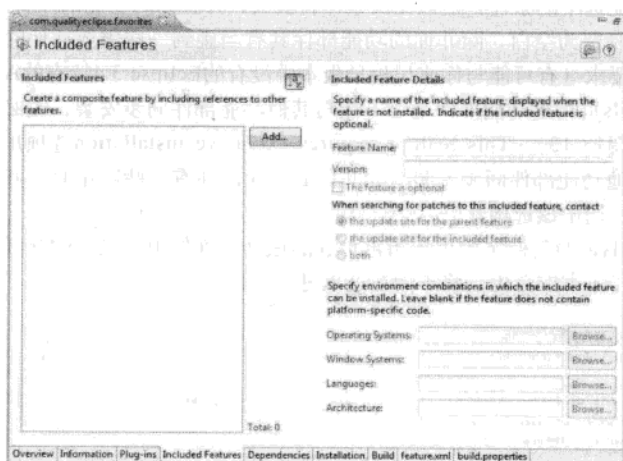


图18-11 包含的功能部件页

可选的子功能部件还可以独立于它们的父功能部件在Update Manager中激活或禁用。Feature Name字段用于为在它丢失事件中的该功能部件提供名称（出于显示目的）。

Dependencies页包含一个所有必须可用以安装你的功能部件的功能部件和插件列表（图18-12）。如果它们中有任何项丢失了，你的功能部件将无法载入。根据稍早的说明，所需插件列表是基于混合由功能部件中插件指定的所需插件而首次生成的。

你可以通过点击Add Plug-in...或Add Feature...按钮手动添加插件或功能部件至列表。点击Compute按钮将根据由功能部件包含的插件所指定的要求重新生成列表。

对于每一个所需插件（可选的）可以指定一个Version to match和一个Match Rule。Match Rule字段中的以下选项控制插件的哪些版本是可接受的先决条件。

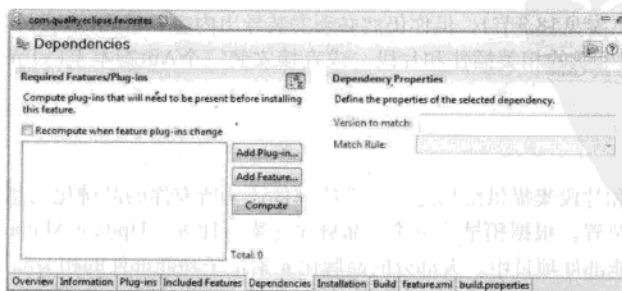


图18-12 依赖项页

- Perfect表示插件必须完全匹配提供的版本号。
- Equivalent表示版本可以在服务或限定符级不同。
- Compatible表示插件可以具有一个比较新的较小版本号。
- Greater or Equal表示插件可以具有任意比较新的版本号。

如果版本号不匹配选中的条件,则该先决条件将丢失,并且你的功能部件将不会载入。一般地,你将很可能将这些字段留为空白,除非你的功能部件具有当碰到一些需要插件的错误版本时将导致它失败的非常特定的要求(有可能与你使用稍早版本中没有的Eclipse 3.4特有的API类似)。

Installation Details页指定功能部件是否不能与其他功能部件同步安装,或必须安装至其他功能部件的同样目录中(图18-13)。This feature requires exclusive installation选项用于阻止你的功能部件被与某个数量的其他功能部件同步安装。除非你的功能部件有一些阻止它与其他功能部件一起安装的特别之处,你应不选中该选项。

Installation Details页的其他字段用于指定可选的该功能部件中应包含的非插件项和高级安装处理器。这些内容超出了本书的范围,将不会在这里进一步讨论。

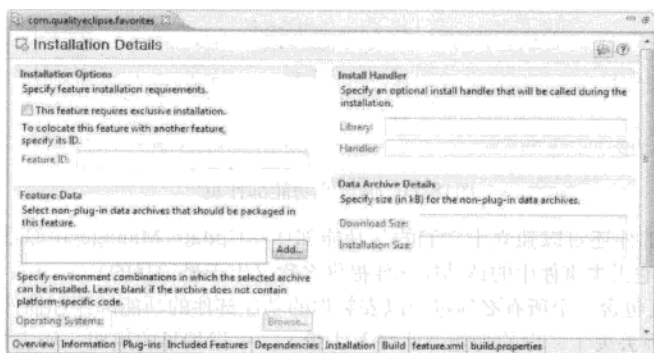


图18-13 安装细节页

18.1.4 测试功能部件

在此时,唯一测试功能部件的方法是使用功能部件清单编辑器中的Export Wizard链接导出它(图18-6)并将它安装至一个Eclipse实例。作为选择,你可以通过执行一个Ant脚本(参见第19章)或创建一个更新站点(参见18.3节),但你仍然必须安装导出内容并重启Eclipse。你可以安装功能部件和由Export Wizard生成的相关插件和片段,或直接安装一个Ant脚本至Eclipse(参见2.5节)或dropins文件夹。

18.2 品牌化

除了向你的插件和片段集提供结构之外,功能部件还为所有你的品牌化信息(比如,About页、图像等)提供了一个位置。根据稍早的讨论,品牌化元素(比如,Update Manager中显示的标题图像)将不会放置于功能部件项目中。大部分的品牌化元素位于功能部件的相关品牌化插件。

此时,有大量不同的品牌化文件。一些仅适用于Eclipse产品(使用Eclipse平台创建的独立程序),

而其他的可以适用于任意功能部件。适用于任意类型的功能部件的文件包括：

- about.html
- about.ini
- about.properties
- about.mappings
- <featureImage> (在about.ini文件中命名)

其他文件，仅适用于特定产品，包括：

- <aboutImage>
- <windowImages>
- plugin_customization.ini
- plugin_customization.properties
- splash.bmp

提示 请一定要编辑build.properties文件，以使这些文件在从工作区中导出功能部件时将包括这些文件（参见18.1.4节）。

18.2.1 about.html文件

每个功能部件和插件应包括一个about.html文件。这是一个简单的HTML文件。它在用户打开Eclipse About对话框，打开Plug-in Details对话框，选择插件，然后点击More Info按钮时显示（图18-14）。

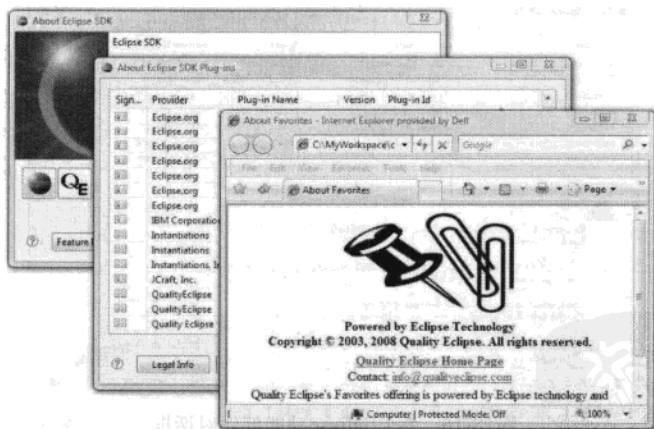


图18-14 收藏夹插件的About页

请注意，为了满足《RFRS requirements》，about.html必须包含以下语句：“This offering is powered by Eclipse technology and includes Eclipse plug-ins that can be installed and used with other Eclipsebased offerings”。

18.2.2 about.ini文件

位于功能部件的品牌化插件中的about.ini文件控制大部分的功能部件品牌信息。它是一个标准

的Java属性文件。它包含了特殊的键（参见表18-1），比如功能部件的关于文本、在Eclipse About对话框中显示的图像的名称（图18-15）等。

第一个键，aboutText，是功能部件的一个简短多行描述。功能部件应提供它的名称、版本号、相关构建信息、版权信息等。文本将会在About Features对话框中显示（图18-16）。可以在About对话框中点击Feature Details按钮访问该文本。

表18-1 about.ini键

键	描 述
aboutText	功能部件的简短的多行描述
featureImage	用于About对话框中的32 × 32像素的图像
tipsAndTricksHref	到提示与技巧帮助页的链接

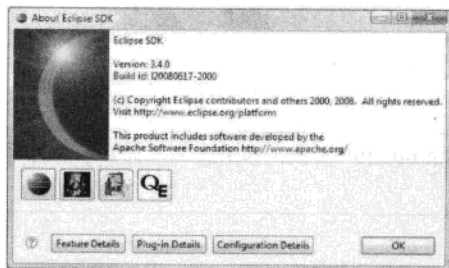


图18-15 Eclipse关于对话框

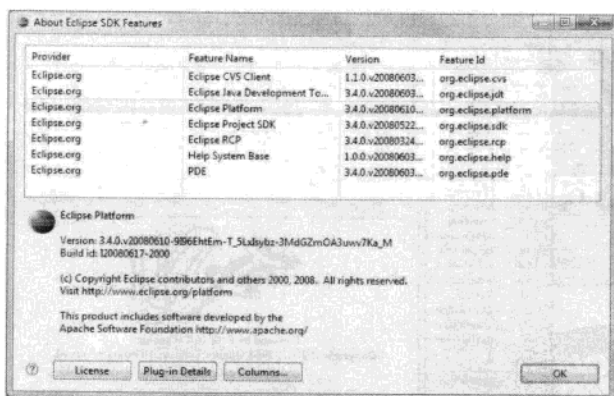


图18-16 关于Eclipse功能部件对话框

为了本地化（翻译）该信息，该文本可以被放置于一个相关的about.properties文件中，并放置一个本地化键于about.ini文件自身。关于文本还可以使用由about.mappings文件提供的值赋予参数。

该内容可以成为有用的编码信息。该信息基于产品版本（如版本号）或产品安装（如用于的名称或许可证号）而改变。比如，关于文本可能包含语句“this product is licensed to {0}”。而“{0}”表示一个参数数字。该参数匹配于对应值，比如，在映射文件中，“0=Joe User”。

about.ini文件中的下一个键，featureImage，用于引用一个32 × 32像素的图像。该图像将用于表示主About对话框中的功能部件（图18-15）和About Features对话框（图18-16）。如果安装了多个功

能部件，那么这些功能部件的图像将会在About对话框的底部一字排开。

如果功能部件的文档包括一个“提示与技巧”部分，那么你可以使用tipsAndTricksHref键引用它。你可以通过选择Help > Tips and Tricks命令访问任意功能部件的提示与技巧。

Favorites功能部件的about.ini文件应最终与以下内容类似：

```
# about.ini
# contains information about a feature
# java.io.Properties file (ISO 8859-1 with "\" escapes)
# "%key" are externalized strings defined in about.properties
# This file does not need to be translated.

# Property "aboutText" contains blurb for About dialog. (translated)
aboutText=%blurb

# Property "featureImage" contains path to feature image. (32x32)
featureImage=feature.gif
```

about.properties文件包含about.ini文件中所有可翻译的字符串。对于Favorites功能部件，该文件应与以下内容类似：

```
# about.properties
# contains externalized strings for about.ini
# java.io.Properties file (ISO 8859-1 with "\" escapes)
# fill-ins are supplied by about.mappings
# This file should be translated.

blurb=Favorites\n\
\n\
Version: 1.0.0\n\
Build id: {0}\n\
\n\
(c) Copyright Quality Eclipse. 2003, 2009. All rights reserved.\n\
Visit http://www.qualityeclipse.com\n\
\n\
This offering is powered by Eclipse technology and includes\n\
Eclipse plug-ins that can be installed and used\n\
with other Eclipse-based offerings.
```

18.2.3 产品品牌化

剩下的品牌化文件仅适用于产品并被指定添加至org.eclipse.core.runtime.products扩展点（图18-17）。作为示例，这里是一个位于org.eclipse.platform插件中的扩展项定义：

```
<extension id="ide" point="org.eclipse.core.runtime.products">
  <product name="%productName"
    application="org.eclipse.ui.ide.workbench"
    description="%productBlurb">
    <property name="windowImage"
      value="eclipse.png,eclipse32.png"/>
    <property name="aboutImage" value="eclipse_lg.png"/>
    <property name="aboutText" value="%productBlurb"/>
    <property name="appName" value="Eclipse"/>
    <property name="preferenceCustomization"
      value="plugin_customization.ini"/>
  </product>
</extension>
```

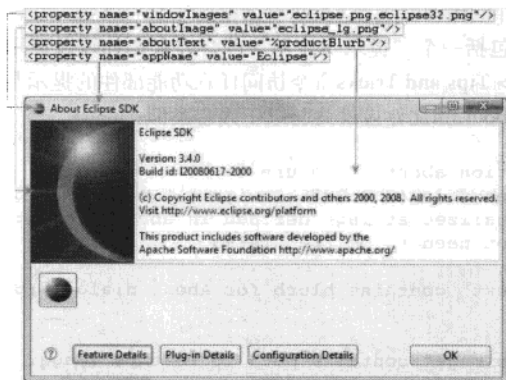


图18-17 产品品牌化属性

windowImages属性指向一个16×16像素的图像。该图像作为窗口和对话框左上角的图标。对于Eclipse工作台自身，这是无处不在的Eclipse图标。

aboutImage属性指向一个更大的图像。该图像被放置于紧挨着主About对话框中的关于文本。如果该图像的大小小于250×330像素，那么它将紧挨着文本显示。如果图像更大（至500×330像素大小），将会压缩关于文本。

根据稍早对about.ini文件的描述，aboutText属性是产品的一个简短多行描述。产品会提供它的名称、版本号、相关构建信息、版权信息等。该文本将在产品About对话框中显示（图18-15）。

appName属性用于为程序提供一个不可翻译的名称。对于Eclipse，即字符串“Eclipse”。

如果产品需要更改任意其他已安装的插件的默认首选项，那么它可以将这些新的设置放置于一个由preferenceCustomization属性指定的文件中（如plugin_customization.ini文件）。该文件中的每一行都应遵循以下样式：

```
<plug-in id>/<preference id>=<value>
```

如果需要本地化任意的值，翻译后的值应放置于plugin_customization.properties文件中。该文件遵循第16章中建立的样式。

产品弹出窗口的位置由产品configuration目录中的config.ini中的osgi.splashPath属性指定。Eclipse将特别根据名称查找的splash.bmp文件，包含产品弹出窗口。它应是一个24位彩色位图文件，并且它的大小应精确为500×330像素。如果弹出窗口中的文本需要被本地化，splash.bmp文件可以被放置于片段中。

18.3 更新站点

在你已经创建了一个功能部件，并为你的插件提供了一个统一结构和商标标识后，你需要向用户分发你的功能部件。你可以将功能部件打包为一个压缩ZIP文件，或创建你自己的安装文件（使用InstalShield或其他类似工具），Eclipse提供了一个有吸引力的基于网络的替代方案。它可以管理分发、安装和你的功能的可用更新。

Eclipse更新站点是一个特别构造的网站。它被设计用于存储你的功能和插件（被打包为JAR文件）并使用一个特殊的站点清单文件对它们进行描述（site.xml文件）。Eclipse Update Manager可以

读取该站点清单文件并自动载入和安装它找到的所有更新（对于新产品而言）。

18.3.1 创建更新站点项目

如同你工作区中的项目所展示的插件、片段和功能部件，更新站点也是一样。为了创建更新站点，我们首先使用New Project向导创建一个新的Update Site Project（图18-18）或点击功能清单编辑器中的Update Site Project链接（图18-6）。

在New Update Site向导的第一也是唯一的一页（图18-19），输入“com.qualityeclipse.favorites.update”作为项目名称。Web Resources选项控制向导是否将为更新站点生成一个默认的主页。如果用户手动访问更新站点，这是他们将要使用的页。选中Generate a sample web page listing all available features within the site单选框并将Web resources location字段设为“web”。点击Finish按钮以创建该项目和它的初始文件。

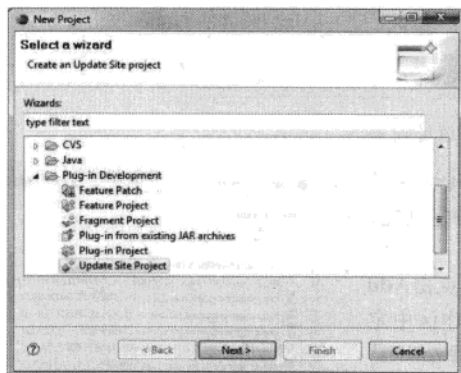


图18-18 新建项目向导——选择更新站点项目

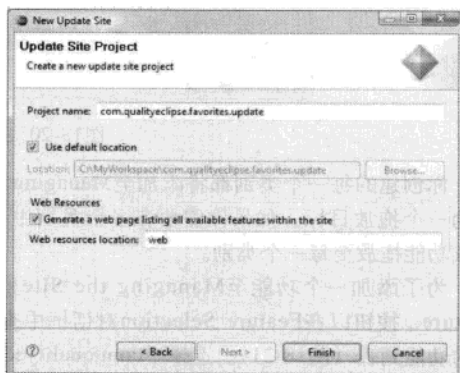


图18-19 新建更新站点向导

默认创建了几个文件和目录：

```
/web
  site.css
  site.xml
index.html
site.xml
```

在你已经添加了一个功能部件至站点后（参见下一节），两个附加目录，/features和/plugins，将保存包含功能部件和插件文件的JAR文件。当这些文件被上传至更新站点后，它们可以通过Update Manager访问。/web目录包含用于提供更新站点外观的样式清单文件。index.html文件是该站点的主页。它的大部分内容是基于更新站点的内容动态创建的。

18.3.2 site.xml文件

最重要的文件是站点清单文件——site.xml。一开始，它是空的，以用于所有实际目的，所以你需要填充它的内容。站点清单编辑器提供了一个方便的接口以用于编辑站点的已有的特性或添加新的属性。

双击站点清单文件将打开站点清单编辑器（图18-20）。该编辑器有三个页面——Site Map、Archives和site.xml。

如果通过更新站点可以使用多个功能，你可能需要将它们分类放置。点击New Category按钮以创建一个新的类别。每个类别应具有唯一的名称（Name）、标签（Label）和将出现于更新站点和Update Manager中的描述（Description）。对于本更新站点，输入“Favorites”作为Name，“Favorites Features”作为Label。

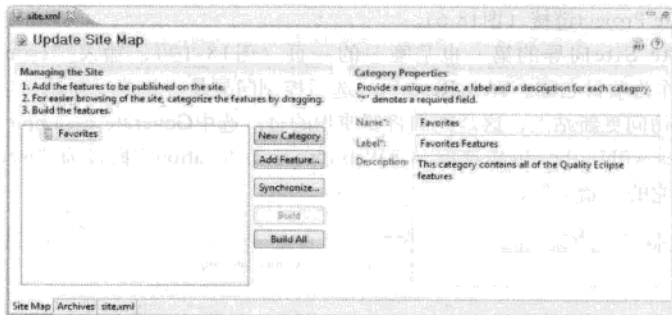


图18-20 站点清单编辑器

你创建的每一个类别都将添加至Managing the Site列表并作为一个拖放目标。如果你需要在多个类别中显示特定功能，将该功能拖放至每一个类别。

为了添加一个功能至Managing the Site列表，点击Add Feature...按钮以在Feature Selection对话框中查看工作区中定义的功能列表（图18-21）。选择“com.qualityeclipse.favorites”功能并点击OK以添加Favorites功能至列表。

点击Managing the Site列表中的任意功能将显示Feature Properties和Feature Environments（图18-22）。你将看到一个必需字段和一些可选字段。这些字段可以用于提供关于功能的细节。必需的（也是无法更改的）URL字段用于指定更新站点的位置（相对于site.xml文件）。Update Manager可以期望在该位置找到功能的JAR文件。对于Favorites功能，该字段应为“features/com.qualityeclipse.favorites_1.0.0.jar”。This feature is a patch for another feature选项指定该功能是否用于修补已有功能（与更新它至一个新的版本相对）。

剩下字段是可选的，并用于指定选中功能对于哪些环境是可用的。它们与你在功能清单编辑器中看到的功能十分类似（图18-6）。你通常将设置这些字段为空白，除非你的功能具有特殊的运行时要求。

在这时点击Build All按钮将添加/features和/plugins目录至项目，并使用包含功能和插件文件的JAR文件填充它们。它还会生成一个p2手动存储索引（artifacts.xml）和元数据存储索引（content.xml）。

```
/features
  com.qualityeclipse.favorites_1.0.0.jar
/plugins
  com.qualityeclipse.favorites_1.0.0.jar
```

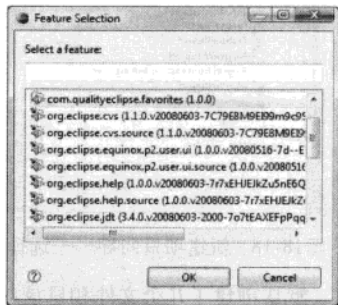


图18-21 功能选择对话框

```

com.qualityeclipse.favorites.help_1.0.0.jar
com.qualityeclipse.favorites.help.n11_1.0.0.jar
com.qualityeclipse.favorites.n11_1.0.0.jar
artifacts.xml
content.xml

```

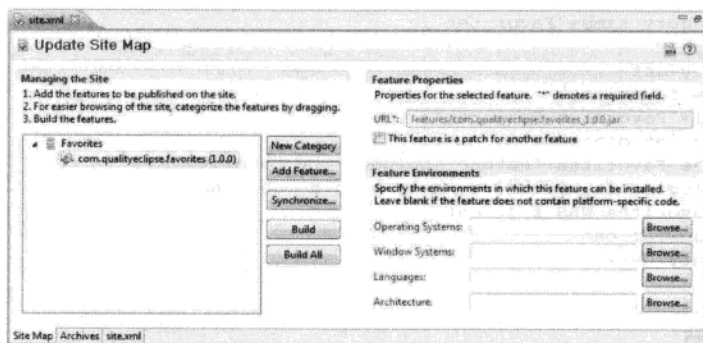


图18-22 显示功能属性的站点清单编辑器

Archives页（图18-23）描述了更新站点并指定它的网址、描述和所有附加数据存档。URL字段包含更新站点的根网址。对于Favorites示例，你将输入“http://com.qualityeclipse.com/update”。最后，在Description字段中输入站点的描述，并将Archive Mapping部分保留为空白。

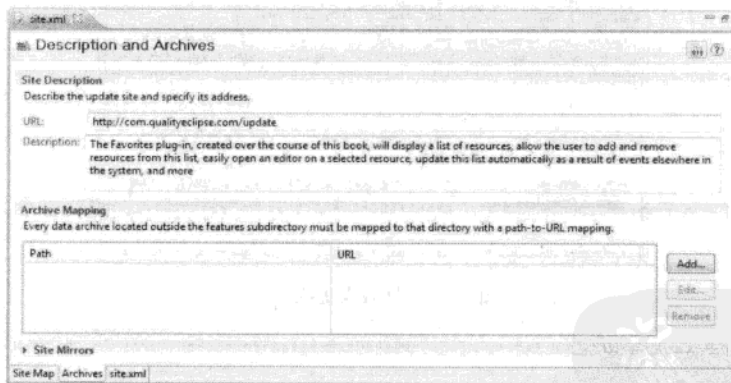


图18-23 站点清单编辑器的存档页

此时，如果你切换至编辑器的site.xml页，site.xml文件的源代码将与以下内容类似：

```

<?xml version="1.0" encoding="UTF-8"?>
<site>
  <description url="http://com.qualityeclipse.com/update">
    The Favorites plug-in, created over the course of this book,
    will display a list of resources, allow the user to add and
    remove resources from this list, easily open an editor on a
    selected resource, update this list automatically as a result

```

```

    of events elsewhere in the system, and more.
</description>
<feature
    url="features/com.qualityeclipse.favorites_1.0.0.jar"
    id="com.qualityeclipse.favorites"
    version="1.0.0">
    <category name="Favorites"/>
</feature>
<category-def
    name="Favorites"
    label="Favorites Features">
    <description>
        The Favorites feature includes the Favorites
        plugin, the Favorites help plugin, and the
        Favorites NLS fragment.
    </description>
</category-def>
</site>

```

18.3.3 更新网站

现在，你可以看到更新站点的样子了。复制不同的站点地图和更新项目内的JAR文件至更新网站——在Favorites示例中是www.qualityeclipse.com/update/。当文件都上传后，你可以用浏览器打开更新站点的URL。Favorites更新站点将显示你在站点清单编辑器中定义的描述、类别和功能（图18-24）。你也可以通过右键点击并选择Open With > Web Browser查看工作台中的index.html文件。

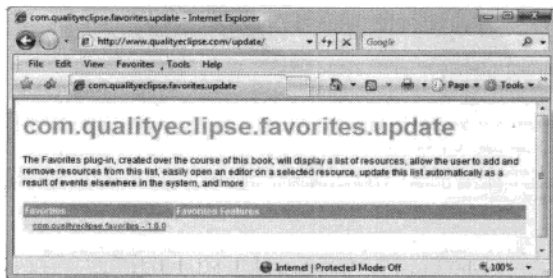


图18-24 收藏夹更新站点

18.3.4 回到功能部件清单

在本章前面第一次讨论功能部件清单文件时，我们跳过了Update Site URL。根据之前的描述，当Update Manager为插件寻找更新时，它将查看由更新URL定义的站点。

重新打开功能清单编辑器（参见18.1.3节），并访问Overview页。在编辑器的General Information部分，更改Update Site URL为“<http://www.qualityeclipse.com/update/>”，Update Site Name为“Quality Eclipse”（图18-25）。在任意

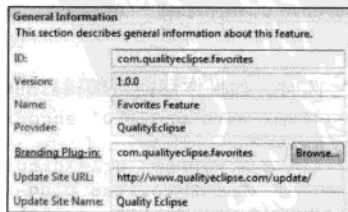


图18-25 功能清单文件中的功能URL

Favorites功能需要检查更新时，它将搜索指定的更新站点。

18.3.5 访问更新站点

可以通过多种方法使用Eclipse Update Manager访问更新站点。选择Help > Software Updates...命令将打开Software Updates and Add-ons对话框（图18-26）。选择Installed Software选项卡将显示工作区中已载入的功能的列表和版本号。

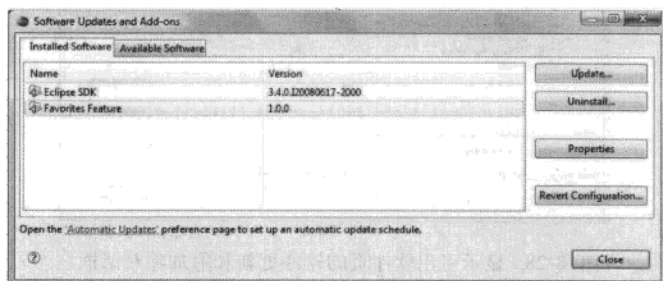


图18-26 显示已安装软件页的软件更新和附加项对话框

对于每个功能，可使用几种不同的任务。Uninstall任务可以用于卸载功能，将使得它的所有添加项从工作区中消失。Properties任务将打开一个属性对话框以显示版本、提供者名称、许可证协议和功能的其他内容（图18-27）。

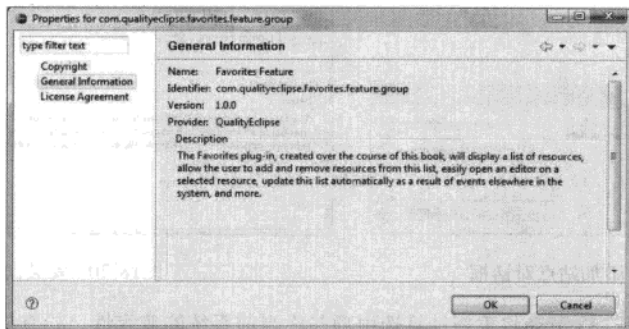


图18-27 收藏夹功能的属性

Eclipse Update Manager还可以用于管理更新站点。选择Software Updates and Add-ons对话框的Available Software选项卡（图18-28）将显示一个所有更新站点列表，以用于查找可用软件。

如果你点击Add Site...按钮并指定一个具有URL为“http://www.qualityeclipse.com/update”的更新站点位置（图18-29），向导将扫描该更新站点，读取站点清单，并自动查找“Favorites Feature”功能（图18-28）。

提示 如果更新站点位于网络上，输入站点的URL至Location文本字段。你也可以从一个浏览器粘贴或拖放URL。如果更新站点位于本地文件系统（比如，你工作区的收藏夹更新项目），

点击Local...按钮以指定站点的目录位置。如果更新站点被打包为一个JAR或zip文件, 点击Archive...按钮以指定文件的名称。

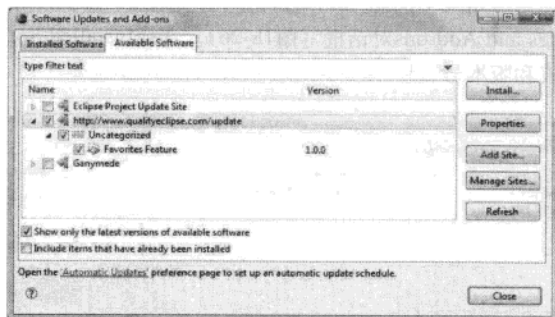


图18-28 显示可用软件页的软件更新和附加项对话框

选择一个更新站点并点击Install...按钮将打开Install向导, 以显示所有发现的功能和版本号。选择一个功能将显示它的大小和细节描述 (图18-30)。

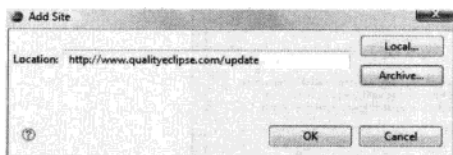


图18-29 添加站点对话框

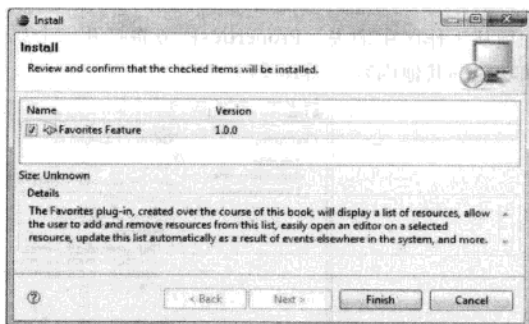


图18-30 安装向导

每次检查一个项, Eclipse将重新计算选中项与你当前系统的兼容性。检查你打算安装的功能并点击Finish按钮以切换至显示功能的许可证的Review License页 (图18-31), 你必须接受该许可证以安装功能。点击Finish按钮将启动安装过程。

请注意 一些项可能被提供者的组织进行了数字签名。这将允许你验证将要下载并安装的功能是来源于一个可信的源。当检测到签名时, 可能提示你验证签名后的内容。

一旦成功下载了所有的软件, 且需要的文件都安装至Eclipse后, 将提示你需要重启Eclipse。当提示退出并重启Eclipse以使更改生效时, 点击Yes。

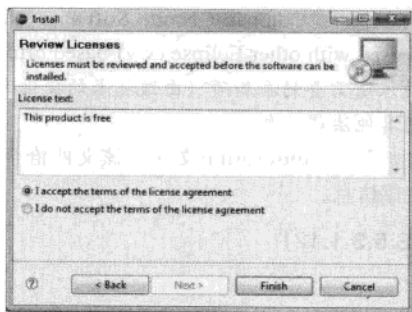


图18-31 显示查看许可证页的安装向导

18.4 RFRS相关事项

《RFRS Requirements》的“功能商标”一节包含了与商标相关的1个要求和4个最佳做法。

18.4.1 不要覆盖产品品牌 (RFRS 3.1.8)

要求#1说明:

产品商标和启动点被限制于已经包括于Eclipse平台中的产品安装。扩展项安装可以通过重写已有的产品配置而不覆盖已有的产品品牌。扩展项安装可以不创建使用不同配置启动平台的快捷方式。如果一个扩展项需要启动Eclipse平台，他们必须安装自己的副本。

为了通过该测试，展示你的功能不覆盖任意已有的产品品牌（通过替代任意文件或使用功能启动选项）。

18.4.2 具有品牌的功能部件可见性 (RFRS 5.3.1.9)

最佳做法#2说明:

当扩展项已经安装并可用时，至少有一个具有品牌的功能必须在About product_name和About product_name Features对话框中显示。商业伙伴应考虑为至少一个功能提供品牌并为所有已安装功能引用的每一个插件提供合适的文档。功能部件品牌内容（当提供了时）必须是完整且正确的。

打开Eclipse About对话框并展示你的功能图标出现于按钮上。然后，打开About Eclipse SDK Features对话框，在列表中选择你的功能，并展示你的功能细节是显示的（图18-15）。

18.4.3 包含添加项信息 (RFRS 5.3.1.10)

最佳做法#3说明:

商业伙伴的功能和插件必须在从About product_name对话框中使用Feature Details...和Plug-in Details...按钮打开的添加项对话框中包含合适的添加项信息（公司名称、版本id、名称）。

展示功能的关于文本（图18-15）包含了公司名称、功能版本ID等。

18.4.4 about.html文件内容 (RFRS 5.3.1.11)

最佳做法#4说明:

插件必须在插件安装目录中包含一个about.html文件。它最少应包含:

- a. Eclipse添加项, 使用以下文本: “Company_Name Software_Name offering includes Eclipse plug-ins that can be installed and used with other Eclipse (x.y)-based offerings.”
- b. 由插件使用的任意独立技术所需要的添加项 (由该技术的提供者定义)。
- c. 所有提供者被授权提供的其他法律信息。

展示你的每一个插件都包含了一个about.html文件。该文件恰当地提到了Eclipse技术的使用(图18-13)以及所有其他相关法律信息。

18.4.5 启动画面限制 (RFRS 5.3.1.12)

最佳做法#5说明:

仅当软件被安装为评估或论证模式时,才允许功能显示启动图像。启动图像的显示可能不干扰用户或需要特定操作以让它消失。当购买了许可证后,软件必须在许可证的应用中自动被修改以移除所有功能特定的启动图像的显示。

对于该测试而言,展示你的功能要么不具有它自己的启动画面,要么在评估时间完成后恰当地取消它自己的启动画面。

18.5 总结

一旦你已经创建了你产品的插件,功能部件提供了一种机制以用于添加结构和品牌。品牌元素,如About页,将与功能部件相关联。Eclipse Update Manager可以载入和卸载被打包为一个功能部件的一组插件,并可以搜索基于网络的更新站点以查找新的版本。

参考文献

本书资源 (2.9节).

Equinox p2 Getting Started (http://wiki.eclipse.org/Equinox_p2_Getting_Started).

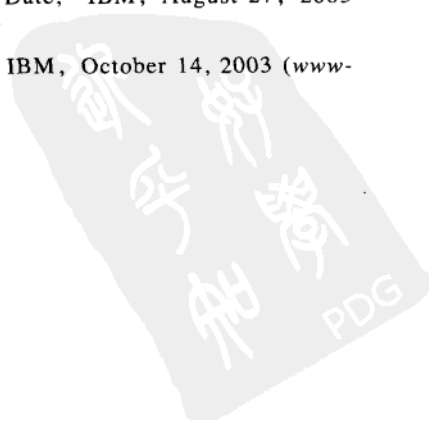
Eidsness, Andrew, and Pascal Rapicault, “Branding Your Application,” OTI, September 16, 2004 (www.eclipse.org/articles/Article-Branding/brandingyour-application.html).

Adams, Greg, “Creating Product Branding,” OTI, November 27, 2001 (www.eclipse.org/articles/product-guide/guide.html).

Glozic, Dejan, and Dorian Birsan, “How to Keep Up to Date,” IBM, August 27, 2003 (www.eclipse.org/articles/Article-Update/keeping-up-to-date.html).

McCarthy, Pat, “Put Eclipse Features to Work for You,” IBM, October 14, 2003 (www-128.ibm.com/developerworks/opensource/library/os-ecfeat/).

Eclipse Help: PDE Guide > Exporting a plug-in



第19章 构建产品

根据2.4节中所介绍的，构建产品包含了仅打包那些将被分发至用户的元素，为一种用户可以安装至其环境的格式。虽然你可以手动构建产品，更好的方法是花费一些时间创建一个更严密的、能在长时间运行中节省时间的自动构建步骤。这一章讨论的正是Favorites产品的这样一个自动构建步骤，并改进2.4.2节中介绍的构建脚本。

19.1 Ant的简要介绍

Ant是Apache网站 (ant.apache.org/) 的一个构建工具。它作为Eclipse的一部分而分发。它与make和它的其他同类不同，因为Ant是完全用Java编写的，可以不使用任意本地系统特定的代码进行扩展，并具有一个基于XML的语法。之后是一个关于Ant和它的语法的非常简单的介绍。为了了解更多信息，参见Ant网站。

19.1.1 构建项目

Ant构建脚本是基于XML，并具有以下结构：

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="target2" basedir=".">
  <target name="target1">
    <task 1a>
    <task 1b>
    ... more tasks here ...
  </target>
  <target name="target2" depends="target1">
    <task 2a>
    <task 2b>
    ... more tasks here ...
  </target>
  ... more targets here ...
</project>
```

每一个Ant构建脚本具有一个项目元素。该项目元素具有以下属性：

- basedir（可选的）——指定将在脚本被执行时使用的工作目录。
- default（可选的）——指定将在脚本运行且未指定目标时将要运行的默认目标。
- name（可选的）——指定项目的可读名称。

为了执行一个构建脚本，在Eclipse视图中选择构建脚本，如Resource Navigator，并选择Run As > Ant Build...命令（参见2.4.2节）。

19.1.2 构建目标

项目包含一个或多个目标，每一个目标可以具有以下属性：

- description（可选的）——任务的描述。如果定义了该属性，那么目标将被认为是外部的（external）；如果没有定义，那么它将是一个内部（internal）目标。

- depends (可选的) —— 该任务依赖的任务的名称的逗号隔开的列表 (参见这一节稍后的讨论)。
- name (可选的) —— 其他任务引用该任务的名称。
- if (可选的) —— 执行该任务所必须设置的属性的名称。比如, 对于一个仅当从Eclipse启动Ant构建脚本时才执行的任务, 那么你可以使用`${eclipse.running}`属性 (参见19.1.4节):

```
<target name="myTarget" if="${eclipse.running}">
  ... do something Eclipse related ...
</target>
```

- unless (可选的) —— 执行该任务必须不设置的属性的名称。比如, 对于一个仅当不是从Eclipse启动的Ant构建脚本时才执行的任务, 那么你可以使用`${eclipse.running}`属性 (参见19.1.4节):

```
<target name="myTarget" unless="${eclipse.running}">
  ... do something non-Eclipse related ...
</target>
```

构建目标可以使用`<antcall>`任务在同一个构建脚本中明确调用其他构建目标, 或者使用`<ant>`在不同的构建脚本中调用。作为替代, 目标可以依赖于另一个目标以获得相似的效果, 但仅在同一个构建脚本中。如果目标A依赖于目标B, 而目标B又依赖于目标C (图19-1), 那么如果你执行目标A, Ant框架将首先执行目标C, 然后是目标B, 最后才是目标A。可以让目标A调用目标B, 目标B调用目标C获得同样的效果。

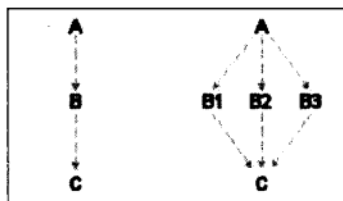


图19-1 构建目标依赖性示例

这两种方法的区别在于目标被执行的次数。比如, 让我们假设一个更复杂的脚本。A依赖于B1、B2和B3, 而这三个都依赖于C。在这种情况下, 如果你执行A, 那么Ant框架将首先执行C, 然后以某种顺序执行B1、B2、B3, 最后是A。请注意, 在这种情况下, 目标C仅被执行一次, 而不是你预期的三次。

19.1.3 构建任务

构建目标由一个将要执行的任务的序列组成。Ant任务有很多不同的类型。一些在随后的三页中列出。在该列表中, 具有以“eclipse”开始的名称的任务将不会被构建为Ant, 而是作为Eclipse的一部分或Eclipse的插件 (参见19.1.7节)。

关于Ant任务的更完整的文档可以在Apache Ant网站 (ant.apache.org/) 和Eclipse帮助中找到。

- ant —— 在另一个构建文件上运行Ant。比如:

```
<ant
  antfile="subproject/subbuild.xml"
  dir="subproject"
  target="compile"/>
```

默认地, 当前构建脚本的所有属性都在当前被调用的构建脚本中可用。作为替代, 你可以设置`inheritAll`属性为`false`, 并且仅用户属性 (比如, 那些通过命令行传递的) 将对于当前被调用的构建脚本可用。在其他情况, 正在被传递的属性集将覆盖被调用的构建脚本中具有同样名称的所有属性。

```
<ant
  antfile="subproject/subbuild.xml"
  dir="subproject"
```

```
inheritAll="false"
target="compile"/>
```

- **antcall**——调用同一个构建文件中的另一个目标（参见19.1.5节）。比如：

```
<antcall target="doSomethingElse">
  <param name="foo.name" value="someValue"/>
</antcall>
```

- **copy**——复制一个或一组由<fileset>指定的文件至一个新的文件或目录。默认地，文件仅当源文件比目标文件要新的时候才被复制，或当目标文件不存在时才被复制。然后，你可以使用overwrite属性明确重写文件。比如：

```
<copy file="myfile.txt" todir="../some/other/dir"/>
```

或

```
<copy todir="../new/dir">
  <fileset dir="src_dir" includes="**/*.java"/>
</copy>
```

上面代码中的<fileset>结构指定了在操作中将被包含的文件。两个星号（**）表示该操作应包含由dir属性指定的目录中的文件，和所有该目录的所有深度的子目录中的文件。

- **delete**——删除一个文件、一个指定目录和所有它的文件和子目录，或一个由<fileset>指定的文件集（参见参考copy以了解更多关于<filesets>的信息）。当指定文件集时，空目录默认不会被移除。为了移除空目录，使用includeEmptyDirs属性。比如：

```
<delete dir="lib"/>
```

或

```
<delete>
  <fileset dir="." includes="**/*.bak"/>
</delete>
```

- **echo**——响应一个消息至当前记录器和监听器，在这里意味着Console视图。比如：

```
<echo message="Hello, world"/>
<echo level="info">Hello, World</echo>
```

- **echoproperties**——响应所有Ant属性至当前记录器和监听器，在这里意味着Console视图。对于调试Ant脚本十分有用。

- **eclipse.buildScript**——生成一个Ant脚本以构建一个指定的插件、片段或功能。

- **eclipse.convertPath**——将一个文件系统路径转换为一个资源路径，反之亦然。将结果分配给指定属性。比如：

```
<eclipse.convertPath
  filePath="${basedir}"
  property="myPath"/>
```

或

```
<eclipse.convertPath
  resourcePath="MyProject/MyFile"
  property="myPath"/>
```

- **eclipse.fetch**——生成一个将从CVS库获取内容的Ant脚本。

- **eclipse.generateFeature**——生成一个将包含指定元素的功能部件。该任务一般被用于从产品配置文件构建RCP程序。

- eclipse.incrementalBuild——触发项目或整个工作区的增量构建，取决于是否指定项目属性。
- eclipse.jarProcessor——签名并打包JAR文件。
- eclipse.refreshLocal——刷新工作区中的指定资源。比如：

```
<eclipse.refreshLocal
  resource="MyProject/MyFolder"
  depth="infinite" />
```

在这里，resource是一个相对于工作区的资源路径，而depth可以是以下参数之一：zero、one或infinite。当Ant构建脚本已经创建，修改或删除Eclipse工作区中的文件或文件夹是十分有用的。Eclipse直到该任务执行时，才会反映工作区中的更改。

- javac——将Java源文件编译为class文件。比如：

```
<javac srcdir="${src}"
  destdir="${build}"
  classpath="xyz.jar"
  debug="on" />
```

- mkdir——创建一个目录，并在需要时，创建它不存在的父目录。比如：

```
<mkdir dir="${dist}" />
```

- property——设置项目中的一个属性或一个属性集（参见19.1.4节）。比如：

```
<property name="foo.dist" value="dist" />
```

或

```
<property file="foo.properties" />
```

- zip——创建一个ZIP文件以包含目录中的一个或多个文件，或由一个<zipfileset>（<zipfileset>与<fileset>类似，参见上面的copy以了解更多关于<fileset>的信息）指定的文件。比如：

```
<zip
  destfile="${dist}/manual.zip"
  basedir="htdocs/manual" />
```

或

```
<zip destfile="${dist}/manual.zip">
  <zipfileset
    dir="htdocs/manual"
    prefix="docs/user-guide" />
</zip>
```

19.1.4 构建属性

属性是一个名/值对。这里的名称是大小写敏感的。属性可以通过将属性名称放置于属性值的“\${”和“}”之间，以用于不同任务属性的值。

```
<property name="builddir" value="c:\build" />
<mkdir dir="${builddir}/temp" />
```

在该构建脚本中，builddir属性在第一个任务中被分配了值“c:\build”，然后，该属性被第二个任务中的dir属性解析，因此创建了c:\build\temp目录。

property任务中的一种替代方案使用location属性：

```
<property name="builddir" location="dir/subdir" />
```

当指定了这种方法时，该值在被关联至builddir属性之前被相对于\${basedir}解析。比如，如果

`${basedir}`是`c:\temp`，那么上面的语句应已经使用值`c:\temp\dir\subdir`与`builddir`关联。如果property任务被稍稍修改（请注意在`dir/subdir`之前添加的斜杠）：

```
<property name="builddir" location="/dir/subdir"/>
```

并且`${basedir}`是`c:\temp`，那么上面的语句应已经使用值`c:\dir\subdir`与`builddir`关联。

提示 使用不具有驱动器名的`location`属性具有更好的可移植性。如果你指定了驱动器名，那么你的构建脚本将只能在Windows平台上运行。

不幸的是，对于未定义属性的引用将不会在Ant执行过程中被报告，而是悄悄地被忽略。如果还没有定义属性，那么不会生成替代字符串。比如，如果在定义`foo`属性之前引用它：

```
<echo message="the foo property is ${foo}"/>
```

那么Ant将保留`${foo}`不被更改，并且显示的消息将是：

```
the foo property is ${foo}
```

这使得发现问题变得更为困难，并且你可能最终碰到一些不常见的文件或目录名，比如：

```
/temp/${plug-in.id}_3.4.0/icons
```

1. 预定义属性

Ant提供了几个预定义属性，包括所有Java系统属性，比如`${os.name}`，和表19-1中展示的内建属性。

表19-1 预定义Ant属性

属 性	描 述
<code>\${basedir}</code>	根据 <code><project></code> 元素的 <code>basedir</code> 属性设置的项目的 <code>basedir</code> 的绝对路径
<code>\${ant.file}</code>	构建文件的绝对路径
<code>\${ant.version}</code>	Ant的版本
<code>\${ant.project.name}</code>	根据 <code><project></code> 元素定义的 <code>name</code> 属性定义的当前执行的项目的名称
<code>\${ant.java.version}</code>	Ant检测到的JVM的版本，如“1.1”、“1.2”、“1.3”、“1.4”、“1.5”或“1.6”

Eclipse提供了五个附加的预定义属性，如表19-2所示。

表19-2 预定义Eclipse Ant属性

属 性	描 述
<code>\${eclipse.home}</code>	Eclipse安装目录的位置
<code>\${eclipse.pdebuild.home}</code>	PDE构建目录的位置（参见19.2.4节）
<code>\${eclipse.pdebuild.scripts}</code>	PDE构建脚本的位置（参见19.2.4节）
<code>\${eclipse.pdebuild.templates}</code>	PDE构建模板的位置（参见19.2.4节）
<code>\${eclipse.running}</code>	如果已经从Eclipse启动了Ant构建，为 <code>true</code> ；如果未定义则为 <code>false</code>

2. 属性范围

属性从它们被声明的那一刻开始在构建脚本中是全局的。如果任务给属性分配一个值，同一个脚本中的另一个任务将也可以使用该属性。在下面的脚本中，`foo`和`bar`属性都是在不同的目标中声明的，并相互引用了对方。

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project name="Test" default="test" basedir=".">
  <target name="init">
    <property name="foo" value="xyz"/>
    <echo message="foo=${foo}"/>
  </target>
  <target name="sub1" depends="init">
    <echo message="foo=${foo}"/>
    <property name="bar" value="abc"/>
    <echo message="bar=${bar}"/>
  </target>
  <target name="sub2" depends="init">
    <echo message="foo=${foo}"/>
    <echo message="bar=${bar}"/>
  </target>
  <target name="test" depends="sub1,sub2">
    <echo message="foo=${foo}"/>
    <echo message="bar=${bar}"/>
  </target>
</project>
```

查看输出，你可以看到属性foo和bar在它们被声明后可以在任何时间被引用。

```
Buildfile: scoping_test_1.xml
init:
    [echo] foo=xyz
sub1:
    [echo] foo=xyz
    [echo] bar=abc
sub2:
    [echo] foo=xyz
    [echo] bar=abc
test:
    [echo] foo=xyz
    [echo] bar=abc
BUILD SUCCESSFUL
Total time: 234 milliseconds
```

对脚本和输出的进一步观察显示了一些令人困扰的内容。bar属性是在目标sub1中声明的，然后在目标sub2中被引用，即使sub2不依赖于sub1。这一点是很重要的，因为Ant不保证非依赖目标的执行顺序。

在这里的第一种情况，目标sub1偶然地在目标sub2之前被执行，因此sub2可以如同预期的那样引用bar属性。如果你根据以下内容修改test目标的depends属性：

```
<target name="test" depends="sub2,sub1">
```

那么sub2目标将在sub1目标之前被执行，导致bar属性在它被引用之后被声明。

```
Buildfile: scoping_test_2.xml
init:
    [echo] foo=xyz
sub2:
    [echo] foo=xyz
    [echo] bar=${bar}
sub1:
    [echo] foo=xyz
    [echo] bar=abc
test:
```

```
[echo] foo=xyz
[echo] bar=abc
BUILD SUCCESSFUL
Total time: 265 milliseconds
```

在简单的测试构建脚本中，问题和解决办法都是显而易见的，但当你的产品（也当你的构建脚本）变得更复杂时，该问题将可能更难被诊断。

提示 底线是当任务A引用任务B中声明的属性时，必须谨慎以保证任务A是直接或间接以来于任务B，以使构建顺序是确定的，并且属性将在它被引用之前声明。

3. 属性可变性

属性一旦声明后就是不可变的。比如，在下面的构建脚本：

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
  <target name="init">
    <property name="foo" value="xyz"/>
    <echo message="foo=${foo}"/>
    <property name="foo" value="123"/>
    <echo message="foo=${foo}"/>
  </target>
  <target name="test" depends="init">
    <echo message="foo=${foo}"/>
    <property name="foo" value="abc"/>
    <echo message="foo=${foo}"/>
  </target>
</project>
```

foo属性在init目标中分配，并且一旦分配后，它不可以被修改（这项规则的特例是<antcall>任务，参见19.1.5节）。不幸的是，多个分配被悄悄地忽略，因此成为了混淆的根源。

Buildfile: mutability_test_1.xml

```
init:
    [echo] foo=xyz
    [echo] foo=xyz

test:
    [echo] foo=xyz
    [echo] foo=xyz

BUILD SUCCESSFUL
Total time: 203 milliseconds
```

4. 目标的外部属性

属性是特殊的，因为它们可以在目标范围之外声明。以这种方式声明的属性在任意目标执行之前定义，并且是不可更改的。比如，在下面的构建脚本中：

```
<project name="Test" default="test" basedir=".">
  <property name="foo" value="xyz"/>
  <target name="test">
    <echo message="foo=${foo}"/>
    <property name="foo" value="abc"/>
    <echo message="foo=${foo}"/>
  </target>
</project>
```

```
</target>
</project>
```

在执行test目标之前分配foo属性的值，它的值不会在测试目标中的第二个property任务中修改。
Buildfile: mutability_test_2.xml

```
test:
    [echo] foo=xyz
    [echo] foo=xyz
BUILD SUCCESSFUL
Total time: 188 milliseconds
```

5. 命令行中的属性

属性还可以在构建脚本外部声明。在命令行声明的属性在启动构建之前定义，并且是不可更改的。

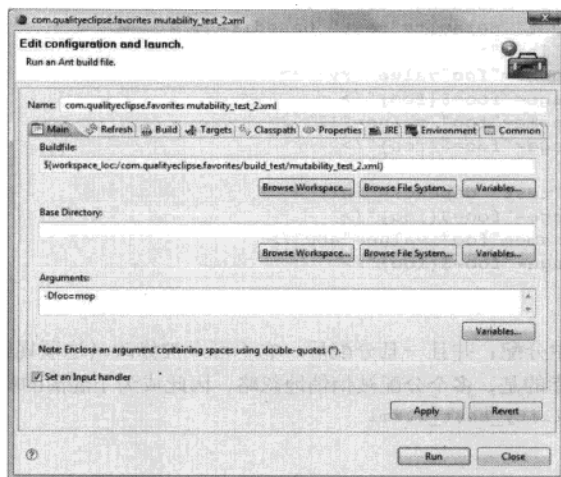


图19-2 声明属性作为Ant命令行的一部分

比如，如果你使用Run As > Ant Build...命令执行上一节描述的构建脚本，切换至Main选项卡面板（图19-2）然后在Arguments字段输入以下内容。

```
-Dfoo=mop
```

那么，将在执行构建脚本之前分配foo属性的值，而且它的值不会由构建脚本内的property声明或property任务所更改。

```
Buildfile: mutability_test_2.xml
test:
    [echo] foo=mop
    [echo] foo=mop
BUILD SUCCESSFUL
Total time: 297 milliseconds
```

作为替代，属性可以通过切换至Properties选项卡面板（图19-3）并不选中Use global properties as specified in the Ant runtime preferences单选框而指定页面顶部包含独立的属性声明，而底部显示了包含属性声明的文件列表。

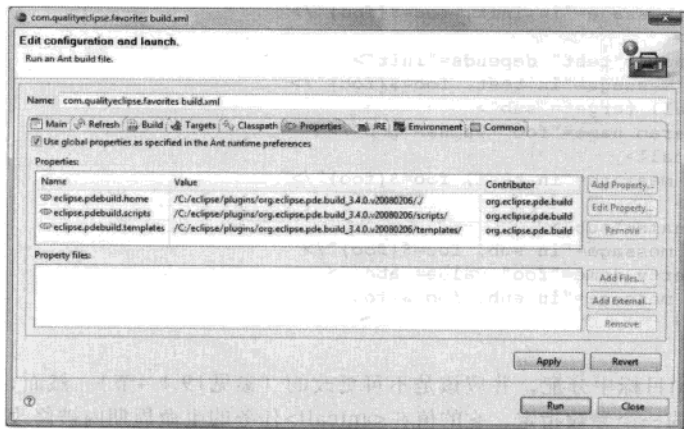


图19-3 声明独立构建脚本可用的属性和属性文件

为了指定适用于工作区中所有构建脚本的属性，打开Eclipse Preferences对话框并浏览至Ant > Runtime首选项页（图19-4）。与前面展示的Properties选项卡面板类似，该首选项页的顶部包含了独立的属性声明，而底部显示包含属性声明的文件列表。

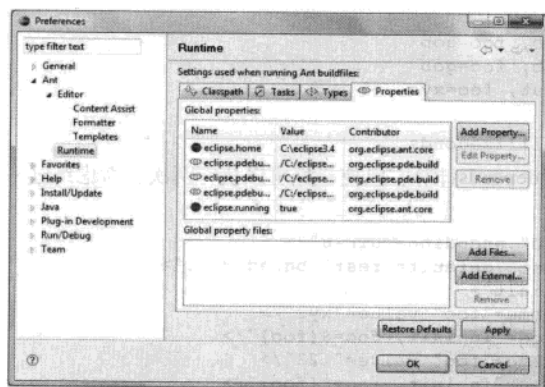


图19-4 声明适用于工作区中所有构建脚本的属性和属性文件

19.1.5 <antcall>任务

<antcall>任务具有一些值得讨论的特别方面。在<antcall>任务中指定的参数覆盖其他地方指定的所有属性。比如，如果执行以下脚本：

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
  <target name="init">
    <property name="foo" value="xyz"/>
    <echo message="in init, foo=${foo}"/>
    <property name="foo" value="123"/>
  </target>
</project>
```



```

    <echo message="in init, foo=${foo}"/>
  </target>
  <target name="test" depends="init">
    <echo message="in test, foo=${foo}"/>
    <antcall target="sub">
      <param name="foo" value="gob"/>
    </antcall>
    <echo message="in test, foo=${foo}"/>
  </target>
  <target name="sub">
    <echo message="in sub, foo=${foo}"/>
    <property name="foo" value="abc"/>
    <echo message="in sub, foo=${foo}"/>
  </target>
</project>

```

foo属性在init目标中分配，并应该是不可更改的（参见19.1.4节）。然而，由于foo是作为<antcall>任务中的一个参数指定，它的值在<antcall>任务的生命周期内被修改。它的原始值在<antcall>任务完成时被保存。

Buildfile: mutability_test_3.xml

init:

[echo] in init, foo=xyz

[echo] in init, foo=xyz

test:

[echo] in test, foo=xyz

sub:

[echo] in sub, foo=gob

[echo] in sub, foo=gob

[echo] in test, foo=xyz

BUILD SUCCESSFUL

Total time: 282 milliseconds

<antcall>任务重设了depends计算，以使目标可以执行两次。考虑经过一个小修改的上一个构建脚本。

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
  <target name="init">
    <property name="foo" value="xyz"/>
    <echo message="in init, foo=${foo}"/>
    <property name="foo" value="123"/>
    <echo message="in init, foo=${foo}"/>
  </target>
  <target name="test" depends="init">
    <echo message="in test, foo=${foo}"/>
    <antcall target="sub">
      <param name="foo" value="gob"/>
    </antcall>
    <echo message="in test, foo=${foo}"/>
  </target>
  <target name="sub" depends="init">
    <echo message="in sub, foo=${foo}"/>
    <property name="foo" value="abc"/>
    <echo message="in sub, foo=${foo}"/>
  </target>
</project>

```



该更改使sub目标依赖于init目标。即使init目标在test目标之前执行，init目标在sub目标之前第二次执行，因为sub目标是使用<antcall>任务执行。此外，foo属性的值与init目标第二次执行时不同，但根据以前的讨论，当<antcall>任务完成时，返回为它的原始值。

```
Buildfile: mutability_test_4.xml
init:
    [echo] in init, foo=xyz
    [echo] in init, foo=xyz
test:
    [echo] in test, foo=xyz
init:
    [echo] in init, foo=gob
    [echo] in init, foo=gob
sub:
    [echo] in sub, foo=gob
    [echo] in sub, foo=gob
    [echo] in test, foo=xyz
BUILD SUCCESSFUL
Total time: 375 milliseconds
```

19.1.6 macrodef

当创建复杂Ant构建脚本时，你将发现相似操作组。重构和参数化这些操作的一种方法是将它们放置于它们各自的目标中，然后通过<antcall>调用它们（参见19.1.5节）。另一种分组并参数化操作的方法是使用macrodef创建新任务。比如，修改上一个示例中的脚本以使用macrodef。

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
  <target name="init">
    <sub fooval="xyz"/>
    <echo message="in init, foo=${foo}"/>
    <sub fooval="123"/>
    <echo message="in init, foo=${foo}"/>
  </target>
  <target name="test" depends="init">
    <echo message="in test, foo=${foo}"/>
    <sub fooval="gob"/>
    <echo message="in test, foo=${foo}"/>
  </target>
  <macrodef name="sub">
    <attribute name="fooval"/>
    <sequential>
      <echo message="in sub,foo=${foo}"/>
      <property name="foo" value="@{fooval}"/>
      <echo message="in sub,foo=${foo}"/>
    </sequential>
  </macrodef>
</project>
```

脚本中要注意的第一件事是对sub macrodef的调用与对内建Ant任务（如echo）的调用十分类似。在该调用中允许的属性通过attribute标签指定。与以下内容类似。

```
<attribute name="fooval"/>
```

如果属性是可选的，并且不需要由调用者指定，那么为属性提供一个默认值。与以下内容类似。

```
<attribute name="fooval" default="a default value"/>
```

属性仅可以在定义它们的macrodef中被引用，并且是“@”而不是“\$”前缀。

- \${foo}——引用构建属性“foo”。
- @{foo}——引用名为“foo”的macrodef属性。

属性根据定义它们的顺序，并在所有构建属性之前解析，从而产生了一些有趣的技术。首先，稍早定义的属性可以被用作稍后定义的属性的默认值。比如，“foo”可以被用作“bar”的默认值。如同下面所示。

```
<attribute name="foo"/>
<attribute name="bar" default="a @{foo} value"/>
```

由于属性（attribute）在性质（property）之前被解析，属性可以用作性质的名称。比如，“foo”属性可以用于指定应将哪一个性质传递至javac任务。

```
<macrodef name="sub">
  <attribute name="foo"/>
  <sequential>
    <javac classpath="${classpath_@{foo}}" ... />
  </sequential>
</macrodef>
```

最后，性质可变性表现为如同sub是一个任务那样。

Buildfile: mutability_test_5.xml

init:

```
[echo] in sub, foo=${foo}
[echo] in sub, foo=xyz
[echo] in init, foo=xyz
[echo] in sub, foo=xyz
[echo] in sub, foo=xyz
[echo] in init, foo=xyz
```

test:

```
[echo] in test, foo=xyz
[echo] in sub, foo=xyz
[echo] in sub, foo=xyz
[echo] in test, foo=xyz
```

BUILD SUCCESSFUL

Total time: 437 milliseconds

19.1.7 Ant扩展项

上面列出的一些任务不是Ant的一部分，而是Eclipse的一部分。表19-3中列出的附加任务将不会在Eclipse外部工作。

表19-3 Eclipse Ant任务提供者

Ant任务	提供者
eclipse.buildScript	作为org.eclipse.pde.build插件的一部分包含于Eclipse
eclipse.convertPath	作为org.eclipse.core.resources插件的一部分包含于Eclipse
eclipse.fetch	作为org.eclipse.pde.build插件的一部分包含于Eclipse
eclipse.generateFeature	作为org.eclipse.pde.build插件的一部分包含于Eclipse
eclipse.incrementalBuild	作为org.eclipse.core.resources插件的一部分包含于Eclipse
eclipse.refreshLocal	作为org.eclipse.core.resources插件的一部分包含于Eclipse

默认地，Eclipse使用一个替代的JRE执行Ant。如果你正使用Eclipse特定的任务，比如以前列出

的那些，那么你将遇到一个与以下列出的类似的错误：

```
Buildfile: com.qualityeclipse.favorites\build.xml
init:
BUILD FAILED: file: com.qualityeclipse.favorites/build.xml:56: Could
not create task or type of type:
eclipsetools_classpath_modifications.

Ant could not find the task or a class this task relies on.
... etc ...
Total time: 406 milliseconds
```

然后你可能需要在与工作区同一个JRE中执行构建脚本。为了完成该任务，右键点击构建脚本，然后选择Run As > Ant Build...。在启动对话框中，选择JRE选项卡并选择Run in the same JRE as the workspace单选按钮（图19-5）。这将使得Eclipse特定的Ant任务可用，以访问底层的Eclipse功能。

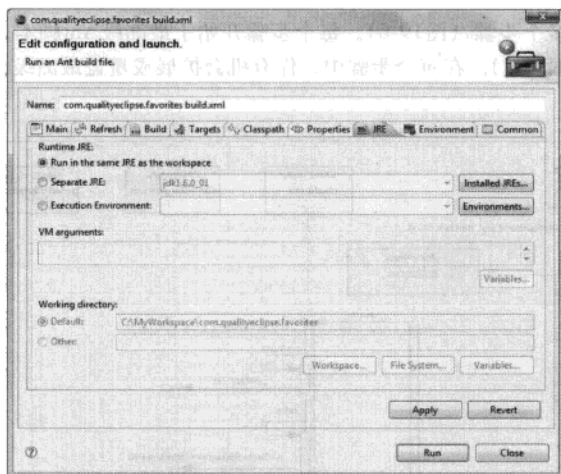


图19-5 Ant启动设置中的JRE选项卡页

19.2 使用PDE构建

现在，让我们重新查看2.4.2节中介绍的Favorites产品构建脚本，并使用一个PDE（Plug-in Development Environment）替换它构建。建立PDE构建可能有一点复杂，但这么做是值得的。一旦完成，构建脚本提供一个可重复的过程以组装你的产品，减轻快速错误修复的难度并使增量改进变得简单。我们在第2章中列出的简单构建脚本对于构建一个插件是合理的，但当使用多个插件构建功能时就不那么好用了。

19.2.1 PDE构建概述

PDE构建过程在不同级别包含了Ant脚本用于构建产品的不同部分：最高级、功能级、插件级Ant脚本。在每个级别，作为构建过程的一部分自动生成了一些Ant脚本，而其他脚本是手动创建的，并位于你的源代码管理系统中。

Ant脚本的子目录：

- features——当你需要附加操作作为构建该功能的一部分而执行时，将被复制至功能项目的Ant脚本（参见19.2.11节）。
- headless-build——可以根据需要复制至你的最高级构建器目录的几个脚本和一个build.properties文件（参见19.2.5节和19.2.11节）。
- packager——用于打包你产品的文件（本书不涉及）。
- plugins——当你需要附加操作作为构建该插件的一部分执行时，将被复制至插件项目的Ant脚本（参见19.2.11节）。

19.2.5 创建PDE构建

产品构建脚本需要放置于某个位置。在我们的情况中，我们创建一个新的com.qualityeclipse.favorites.pde项目以包含它们。我们首先创建该新项目并从上一节提到的templates/headless-build目录复制build.properties文件至该新项目。然后，在该项目中创建一个新的build-favorites.xml Ant脚本，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Build Favorites" default="runBuild">

    <!-- Add content here as described below -->
</project>
```

我们的构建脚本调用main PDE构建脚本。该脚本然后从我们的构建项目中的build.properties文件读取属性。在导入PDE build.xml构建脚本文件之前，我们不直接修改build.properties文件，而是通过在build-favorites.xml Ant脚本中设置它们以覆盖特定属性。

```
<property name="builder" location="." />
<property name="topLevelElementId"
    value="com.qualityeclipse.favorites" />
<property name="baseLocation" location="${eclipse.home}" />
<property name="base" location="${baseLocation}/.." />
<property name="buildDirectory" location="/temp/eclipse.build" />

<import file="${eclipse.pdebuild.scripts}/build.xml" />
```

除了设置19.2.3节中列出的属性之外，上面的代码还设置了以下属性：

- base——baseLocation的父部件
- topLevelElementId——正被构建的功能的标识符

PDE构建过程包括了目标以从CVS中获取构建映射和源代码。这里不使用或讨论这些目标。作为替代，在调用main PDE构建脚本之前，我们必须从工作区复制功能和插件至buildDirectory。

```
<target name="copyProjects">
    <delete dir="${buildDirectory}" />
    <mkdir dir="${buildDirectory}" />

    <record name="${buildDirectory}/${topLevelElementId}-build.log"
        loglevel="verbose"/>
    <echoproperties />

    <mkdir
        dir="${buildDirectory}/features/com.qualityeclipse.favorites" />
```

```
<copy
  todir="${buildDirectory}/features/com.qualityeclipse.favorites">
  <fileset dir="../../com.qualityeclipse.favorites.feature" />
</copy>
<mkdir dir="${buildDirectory}/plugins" />
<copy todir="${buildDirectory}/plugins">
  <fileset dir="..">
    <include name="com.qualityeclipse.favorites/**" />
    <include name="com.qualityeclipse.favorites.help/**" />
    <include name="com.qualityeclipse.favorites.help.nl1/**" />
    <include name="com.qualityeclipse.favorites.nl1/**" />
    <exclude name="*/bin/**" />
  </fileset>
</copy>
</target>
```

提示 上面的Ant目标包括了<record ...>和<echoproperties/>以捕捉输出至文件并分别显示所有Ant属性。这在调试Ant脚本时是十分有用的。

最后,添加runBuild目标。该目标首先复制项目,然后触发main PDE构建脚本。

```
<target name="runBuild" depends="copyProjects, main" />
```

19.2.6 指定编译级别

我们的Favorites产品使用了范型 (generics), 因此是基于Java 1.5的。但默认的编译源代码级别取决于当前使用的Java编译器。解决该问题的一个方法是添加以下属性声明至上一节描述的最高级build-favorites.xml Ant脚本, 明确指定java编译器源和目标级别:

```
<property name="javacSource" value="1.5"/>
<property name="javacTarget" value="1.5"/>
```

作为替代, 我们也可以通过添加以下内容至插件的build.properties文件为该插件设置java编译器源和目标级别:

```
javacSource = 1.5
javacTarget = 1.5
```

这种最高级的方法可以工作, 但阻止了独立插件指定不同的编译级别。你可以通过将这些设置下放至每一个正构建插件的build.properties文件以解决这一问题 (参见19.2.10节)。

一种更好的办法是定义由插件指定的执行环境属性。每个插件可以 (也应当) 在插件清单文件 (参见2.3.1节) 中指定BundleRequiredExecutionEnvironment。定义执行环境属性允许PDE构建过程恰当地关联编译级别和不同的正构建的插件:

```
J2SE-1.4 = <boot.class.path.for.Java.1.4>
J2SE-1.5 = <boot.class.path.for.Java.1.5>
J2SE-1.6 = <boot.class.path.for.Java.1.6>
```

在示例中, 我们为每一个插件指定了J2SE-1.5并使用一个Java 1.5安装运行PDE构建过程, 所以我们添加以下内容至最高级build-favorites.xml Ant脚本:

```
<property name="J2SE-1.5" location="${sun.boot.class.path}" />
```

19.2.7 运行PDE构建

为了构建Favorites产品, 右键点击build-favorites.xml Ant脚本并选择Run As > Ant Build。运行

上面描述的Favorites构建脚本将生成一个包含Favorites功能和插件于构建输出目录的zip文件（参见以下内容）。为了部署Favorites产品，将该文件解压至<eclipse>/dropins文件夹并重启Eclipse。

```
/temp/eclipse.build/I.TestBuild/com.qualityeclipse.favorites-TestBuild.zip
```

提示 该Ant脚本使用了Eclipse特定的Ant任务，如eclipse.buildScript，所以你必须选择Run in the same JRE as the workspace选项以使Ant脚本正确地执行。

你还可以从命令行启动PDE构建过程。为了构建Favorites产品，使用以下内容创建一个build-favorites.bat文件（==>表示上一行的延续，并且不能包含该“==>”）：

```
SET eclipseDir=C:\eclipse

SET eclipseLauncher=%eclipseDir%\plugins\
==> org.eclipse.equinox.launcher_1.0.100.v20080509-1800.jar

java -jar %eclipseLauncher%
==> -application org.eclipse.ant.core.antRunner
==> -buildfile build-favorites.xml
==> 2>&1 1>build.log
```

在上面的代码中，你将需要调整eclipseDir变量以指向你的Eclipse安装和eclipseLauncher变量以指向Eclipse安装内的equinox启动器JAR。代码最后一行末尾的2>&1 1>build.log表达式将java标准输出和标准错误流重定向至当前工作目录的build.log文件。将完整的构建日志捕获至一个文件在调试Ant构建时是十分有用的。

Exception in thread "main" java.lang.NoClassDefFoundError: 0jar 如果你看到该错误消息（请注意u上面的小帽子），那么-jar参数中的横杠可能不是英文的横杠。Word软件（以及Outlook）有时候使用特殊字符替代普通旧式ASCII文本。使用记事本并重新输入该行以解决该问题。（来源于[http://swik.net/Rails-Ruby/TechKnow+Zenze/Debugging+ Users+and+ Invisible+Characters/bza8c](http://swik.net/Rails-Ruby/TechKnow+Zenze/Debugging+Users+and+Invisible+Characters/bza8c)）

如果当前正构建的功能部件和插件还安装于由base.location属性引用的Eclipse安装位置中，那么你可能在PDE构建日志中看到与以下类似的内容（参见以下内容）。该内容表示插件可能没有正确构建。该问题很难被发现，因为构建没有失败，而唯一发现问题的地方位于构建日志的generateScript部分的隐式引用。该引用表示选中了一个具有同样标识符的不同插件。

```
generateScript :
[eclipse.buildScript] Some inter-plugin dependencies
                        have not been satisfied.
[eclipse.buildScript] Bundle com.qualityeclipse.favorites:
[eclipse.buildScript] Another singleton version selected :
                        com.qualityeclipse.favorites_1.0.0
[eclipse.buildScript] Bundle com.qualityeclipse.favorites.n11:
[eclipse.buildScript] Bundle com.qualityeclipse.favorites.help.n11:
[eclipse.buildScript] Bundle com.qualityeclipse.favorites.help:
[eclipse.buildScript] Another singleton version selected :
                        com.qualityeclipse.favorites.help_1.0.0
```

19.2.8 自动生成版本限定符

PDE构建脚本可以为正组装的功能和插件自动生成版本限定符（参见3.3.1节以了解更多关于版

本号的内容)。版本限定符可以表示构建类型并一般包含构建数目或构建日期。在我们的情况中, 我们需要Favorites功能和插件版本限定符作为构建日期, 并包含年、月、日、小时和分。

为了强制在所有当前构建的功能和插件上生成同样的版本限定符, 设置forceContextQualifier属性与想要的版本限定符一致。使用tstamp Ant任务以设置与当前日期和时间相等的版本限定符。在build-favorites.xml Ant脚本中, 在import任务之前, 添加以下内容:

```
<tstamp>
  <format property="forceContextQualifier" pattern="yyyyMMddHHmm" />
</tstamp>
```

然后, 在每一个正在构建的功能和插件的版本末尾添加“.qualifier”。在我们的情况中, 更改功能版本和每一个插件版本为:

```
1.0.0.qualifier
```

当PDE构建运行时, 所有“qualifier”的出现被forceContextQualifier属性的值所替代。

19.2.9 保持版本同步

在产品的发展过程中, 插件的版本必然会随着时间而缓慢地变化。由于feature.xml文件包含对插件和它的版本的引用, 你必须记得在所有你更改插件版本时更新feature.xml文件。为了减少这种情况的发生, 修改feature.xml文件以包含0.0.0而不是每一个被引用插件的实际版本号。在PDE构建过程中, 在feature.xml文件中所有发现的0.0.0将使用该插件清单文件中定义的实际版本所替代。

19.2.10 构建属性

每个功能部件和插件项目都包含一个build.properties文件以描述将包含在构建中的不同的文件和目录。该文件是方便的, 因为Eclipse提供了一个很好的Build Configuration编辑器(图19-8)。该编辑器也作为功能和插件清单编辑器的一部分。在每一个被构建的手动项的PDE构建过程中读取了不同的build.properties文件:

- 最高级——在\${builder}目录的build.properties控制总体的构建过程, 包括正构建的配置、其他文件夹的位置、档案格式, 读取等。
- 功能级——每个功能项目中的build.properties文件指定在功能部署中将包含的内容。
- 插件级——每个插件项目中的build.properties文件指定类似于在插件部署中将包含的源代码位置和附加文件等内容。此外, 你也可以通过添加以下内容至插件的build.properties文件为该插件设置java编译器源和目标级:

```
javacSource = 1.5
javacTarget = 1.5
```

作为替代, 我们可以通过在最高级PDE构建脚本中定义属性为所有插件设置java编译器源和目标级别(参见19.2.6节)。

提示 我们不建议选中图19-8中显示的Custom Build单选框, 并提供你自己的build.xml文件。如果你需要自定义构建操作, 使用PDE自定义回调作为替代(参见19.2.11节)。

切换至编辑器的build.properties选项卡将显示由编辑器操作的属性。在这里, source.favorites.jar属性是一个逗号隔开的favorites.jar的源目录的列表。而output.favorites.jar属性是一个逗号隔开的目录列表。它包含favorites.jar的二进制文件。而bin.includes属性是一个逗号隔开的将在插件自身中包含的文件和目录列表。

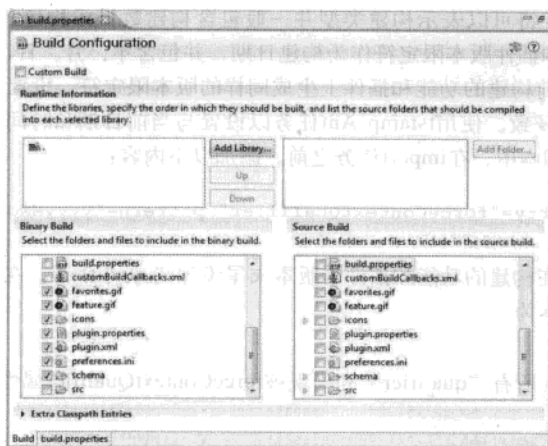


图19-8 构建配置编辑器

```
source.. = src/
output.. = bin/
bin.includes = plugin.xml,\
               META-INF/,\
               .,\
               icons/,\
               preferences.ini,\
               plugin.properties,\
               about.html,\
               about.ini,\
               about.mappings,\
               about.properties,\
               favorites.gif,\
               feature.gif,\
               schema/
```

可以出现于该文件的属性包括：

- `source.<library>`或`source..`——一个逗号隔开的将在编译`<library>`时包含的文件和目录列表。一般地，它是由“.”表示的项目的根或“src/”目录。
- `extra.<library>`——当编译`<library>`但不是在库自身时，将被在classpath中包含的文件和目录的逗号隔开的列表。
- `jars.extra.classpath`——一个逗号隔开的在编译插件源文件时将被包含于classpath的JAR的列表。
- `javacSource`和`javacTarget`——非强制地为java编译器指定源和目标级别（参见19.2.6节中的讨论）。
- `output.<library>`或`output..`——在`source.<library>`中指定的Eclipse将编译文件至的目录。一般地，它是由“.”表示的项目根或“bin/”目录。
- `bin.includes`——一个由逗号隔开的将被包含于插件、片段或功能的文件和目录的列表。

`bin.includes`一般包含对于所有版本的Eclipse都相同的元素。比如，不同Favorites产品项目的`bin.includes`具有以下内容：

- 由Favorites视图和模型使用的图标（参见第7章）

- preferences.ini (参见16.2节)
- 模式文件 (参见17.2.2节)
- META-INF/MANIFEST.MF、plugin.xml和feature.xml文件
- 整个Favorites功能插件和相关文件 (参见第18章)
- 整个Favorites帮助插件 (参见第15章)

19.2.11 自定义PDE目标

你如何自定义PDE构建过程？功能和插件的build.properties文件 (参见19.2.10节) 提供了一些控制方法，比如，使用jars.extra.classpath属性添加附加条目至classpath，但有时候你可能需要更多。以下是一个关于如何添加自定义操作至PDE构建过程的最高级、功能级和插件级的描述。

1. 最高级自定义目标

build-favorites.xml Ant脚本 (参见19.2.5节) 提供了在主PDE构建过程之前和之后的控制方法。为了在PDE构建中的最高级步骤之前或之后添加自定义操作 (参见19.2.2节)，从PDE插件的templates/headless-build目录 (参见19.2.4节) 复制customTargets.xml文件至你的\${builder}目录，并根据需要对它进行修改。一些需要注意的目标包括：

- preSetup——在发生其他任何事情之前调用该目标。
- getMapFiles——仅当skipMaps属性在最高级build.properties文件中被注释时才调用该目标。当可用时，customTargets.xml中的默认实现从\${builder}/maps目录读取映射文件以确定从CVS中读取哪一个功能和插件项目 (参见19.2.2节中的preBuild步骤)。
- preFetch——在从CVS读取源文件之前调用该目标，但仅当skipFetch属性在build.properties文件中被注释时。这在读取之前调整映射文件是有用的。
- postFetch——在从CVS读取源文件之后调用该目标，但仅当skipFetch属性在build.properties文件中被注释时。这在构建继续之前预处理读取的源文件、为CVS添加标签等时有用。
- preGenerate——在为每个正被构建的功能和插件生成build.xml脚本之前调用该目标。
- postGenerate——在为每个正构建的功能和插件生成build.xml脚本之后调用该目标。默认地，该目标执行一个“清理”以移除正被构建的库和插件JAR。最好是在开始构建之前删除整个\${buildDirectory}目录，以确保没有更高级别的手动项影响当前构建。
- preProcess——在编译开始之前调用该目标。
- postProcess——在编译完成之后调用的该目标在通过html或email检查编译日志并报告错误时是有用的。
- postBuild——在PDE构建过程的末尾调用该目标。默认实现收集日志至\${buildDirectory}/\${buildLabel}/compilelogs目录。你也可以运行测试、发布结果、发送通知等。

2. 功能级自定义目标

PDE构建过程在功能级提供了与最高级类似的回调。不实现你自己的功能级build.xml文件，而采取这两个步骤：

- 修改功能的build.properties文件以包含这些行：
`customBuildCallbacks = customBuildCallbacks.xml`
`customBuildCallbacks.failonerror = true`

- 从PDE插件的templates/features目录复制customBuildCallbacks.xml文件 (参见19.2.3节) 至你

功能的项目目录并在需要时对它进行修改

3. 插件级自定义目标

除了最高级和功能级回调之外，PDE构建过程还提供了插件级回调。不实现你自己的插件级build.xml文件，而采取以下两个步骤：

- 修改插件的build.properties文件以包含以下行：

```
customBuildCallbacks = customBuildCallbacks.xml
customBuildCallbacks.failonerror = true
```

- 从PDE插件的templates/plugins目录复制customBuildCallbacks.xml文件（参见19.2.3节）至你插件的项目目录并在需要时对它进行更改。

在插件级还有一些有趣的回调目标：

- pre.build.jars——在该特殊的插件和它的库JAR编译发生之前调用该目标。
- pre.@dot——在为JAR格式的插件中包含的类编译源代码之前调用该目标。
- post.@dot——在为JAR格式的插件中包含的类编译源代码之后调用该目标。
- post.build.jars——在该特殊的插件和它的库JAR编译发生之后调用该目标。
- post.clean——在“清理”过程中调用该目标，并且在删除与该插件关联的，并未被清理的临时构建文件时是有用的。

19.2.12 使用不同版本的Eclipse编辑

有时候，不同的开发者打算在同一个项目上使用不同版本的Eclipse。如果该项目不包含任意的Eclipse插件，那么不会有问题。当项目使用ECLIPSE_HOME classpath变量时就会出现问题。该变量由Eclipse自动管理以指向当前Eclipse安装位置。作为结果，所有使用ECLIPSE_HOME的项目引用当前Eclipse安装位置中的插件。

比如，一位开发者使用Eclipse 3.4将根据Eclipse 3.4插件对项目进行编译，而其他使用Rational Application Developer 7.0，而它是基于Eclipse 3.2的，将根据Eclipse 3.2插件编译该项目。

一种解决办法是使用PDE Target Platform首选项页（图19-9）。使用该页，你可以将ECLIPSE_HOME classpath变量重定向至不同的Eclipse安装目录。使用该方法的问题在于它没有解决使用不同版本的Eclipse编译不同的项目的问题。使用该解决办法，你将根据同一个Eclipse安装编译所有项目。

另一种解决的办法是从不使用ECLIPSE_HOME classpath变量。假设你打算支持Eclipse 3.2、3.3和3.4，安装这三个Eclipse版本至不同的目录，并将它们命名为类似于c:\eclipse3.2、c:\eclipse3.3和c:\eclipse3.4。

然后，建立三个classpath变量，名为ECLIPSE32_HOME、ECLIPSE33_HOME和ECLIPSE34_HOME，分别指向它们各自的Eclipse安装。如果一个项目已经根据Eclipse 3.4进行了编译，你应使用ECLIPSE34_HOME而不是ECLIPSE_HOME。

使用这种方法，使用哪个版本的Eclipse就不那么重要了，因为代码将总是根据一个特定Eclipse版本进行编译。这种方法的缺点是PDE不会保持插件清单依赖项列表与项目classpath文件的同步。

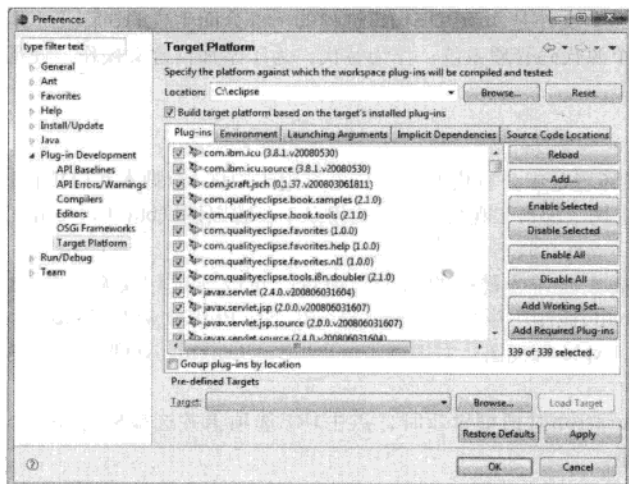


图19-9 PDE目标平台首选项页

19.3 调试PDE构建过程

PDE构建过程可能变得十分复杂，并且任何人在它上面花费一点时间都会发现他们是在调试Ant脚本。PDE UI提供了一些有用的工具，包括Ant脚本生成和调试能力。

19.3.1 自动生成的构建脚本

在PDE构建过程中，为每个正编译和组装的功能部件和插件都生成了一个build.xml文件。为了帮助理解和调试PDE构建过程，PDE Tools可以生成与在PDE构建过程中生成的相同的build.xml文件。在Navigator视图中，右键点击feature.xml或plugin.xml文件并选择PDE Tools > Create Ant Build File。

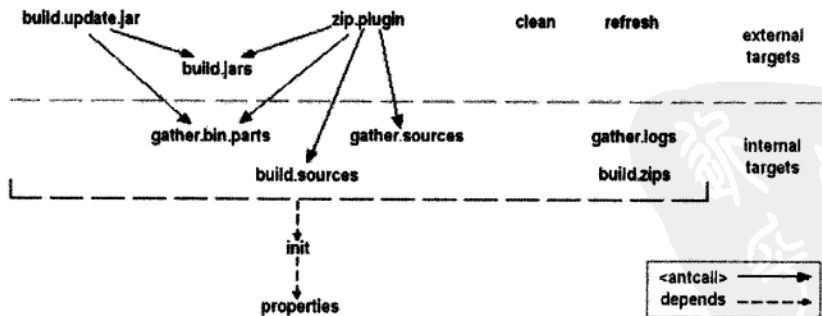


图19-10 PDE生成的构建脚本

这些自动生成的构建脚本有一些复杂。它们包含了5个外部目标和7个内部目标（图19-10）。zip.plugin和build.update.jar目标调用其他目标，而所有目标依赖于init目标。而init目标又依赖于

properties目标。因为这些文件是作为PDE构建过程的一部分而生成的，你将不用手工编辑它们，并且它们不应被导入某个源代码管理系统。作为替代，为了添加自定义操作，使用PDE构建回调（参见19.2.11节）。

19.3.2 使用调试器

Eclipse Java调试环境是十分有用的，并且也有用于调试Ant脚本和PDE构建过程的同样功能。你可以设置断点，步进调试脚本和观察变量。使用以下步骤以调试build-favorites.xml Ant脚本：

- 必须使用Java 1.5或更高版本。
- 导入PDE构建插件以作为链接的二进制项目。可以通过切换至Plug-ins视图，右键点击org.eclipse.pde.build插件，并选择Import As > Binary Project with Linked Content实现。
- 切换回Package Explorer视图，右键点击buildfavorites.xml Ant脚本，并选择Debug As > Ant Build...。
- 当Edit Configuration对话框显示时，点击JRE选项卡并选择Run in the same JRE as the workspace单选框。
- 点击Debug以执行Ant脚本（图19-11）。

你可能注意到偶尔出现的异常情况，比如回调中的断点没有停止执行。如果一个断点看起来没有工作，尝试在更高级别的文件中设置断点，或在最高级构建脚本中设置断点，然后向下步进至回调。

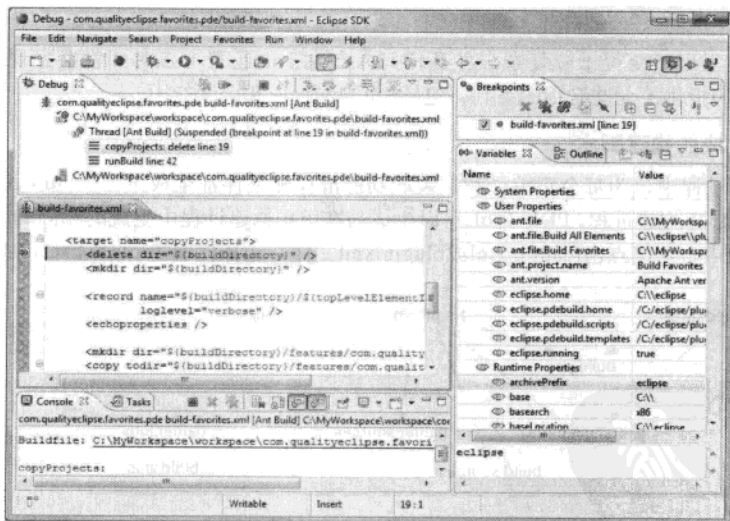


图19-11 调试PDE构建脚本

19.4 总结

你可以手动构建产品，但这样可能会带来错误和构建之前难以察觉的区别。创建一个只需要一次点击、可重复的构建过程对于分发（和重新分发）Eclipse插件是十分必要的。这一章介绍了Ant，然后从细节讨论了如何创建基于Ant的构建过程。

参考文献

本书资源 (2.9节).

PDE Build (<http://wiki.eclipse.org/PDE/Build>).

Pascal Rapicault, Andrew Niefer, Sonia Dimitrov, "PDE Build and build clinic", EclipseCon 2007.

Equinox / p2 (<http://wiki.eclipse.org/Equinox/p2>).

Andrew Niefer, "Example Headless build for a RCP product with p2" (<http://anieferr.blogspot.com/2008/06/example-headless-build-for-rcp-product.html>).

Patrick Paulin, "Getting started with PDE Build" (<http://rcpquickstart.com/2007/06/06/getting-started-with-pde-build>).

Ryan Slobojan, "An in-depth look at Equinox p2 (Provisioning Platform)" (<http://www.infoq.com/news/2008/06/eclipse-ganymede-p2>).

"Equinox p2 Metadata Generator" (http://wiki.eclipse.org/Equinox_p2_Metadata_Generator).

Wassim Melhem and Dejan Glozic, "PDE Does Plug-ins", (<http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html>).

Markus Barchfeld, "Build and Test Automation for plug-ins and features" (<http://www.eclipse.org/articles/Article-PDE-Automation/automation.html>).

Ant Web site (ant.apache.org/).

Eclipse Ant FAQ (eclipsewiki.editme.com/ANTFAQ).

Loughran, Steve, "Ant in Anger: Using Apache Ant in a Production Development System," November 9, 2002 (ant.apache.org/ant_in_anger.html).

QualityEclipse Web site (www.qualityeclipse.com/ant).

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*, Addison-Wesley, Boston, 2005.



第20章 GEF：图形编辑框架

对于许多程序而言，基于文本的编辑是允许用户访问当前由程序修改的数据的最有效方法。然而，其他模型使用一些可视化表示进行最好地显示和编辑。这是图形编辑框架（Graphical Editing Framework, GEF）出现的原因。作为Eclipse项目的GEF，提供了用于对信息进行图形表示的一个开发框架。规则在图形表示上分层排列，以使用户交互修改图形从而更改底层模型。

20.1 GEF体系结构

如同其他显示图形信息的程序一样，GEF框架设计为使用模型-视图-控制器（Model-View-Controller, MVC）框架。MVC由三个主要部分组成：模型、视图和控制器。模型保存将显示的信息，并在会话间得以保存。视图在屏幕上呈现信息，并提供基本的用户交互。控制器协同模型和视图的活动，并在模型和视图间根据需要传递信息。

模型 <-> 控制器 <-> 视图

当向GEF模型添加对象时，将初始化一个对应的控制器。该控制器随后创建表示该模型对象的视图对象。在相反方向，当用户与视图交互时，控制器使用新信息更新模型。

- model——正以图形显示的底层对象（参见20.2节）
- view——包含一个图案集合的GEF画布（参见20.4节）
- controller——一个GEF编辑部分的集合（参见20.3节）

GEF提供了不同的编辑部分（控制器）和图案（视图）类，由用户扩展，以减轻在你的程序中使用图形框架的难度。这一章将使用收藏夹模型（参见7.2.3节）创建GEF视图和编辑器以讨论这些类和GEF框架。在该示例中，将为每一个FavoritesManager中的收藏夹项生成一个图案。该FavoritesManager将指向一个表示收藏夹所引用的资源的图案。GEF的每一个部分（模型、视图和控制器）将在集成至Eclipse之前被单独讨论。

GEF不是作为标准Eclipse SDK的一部分分发，所以在继续之前我们必须安装它。GEF项目由三个子部分组成：

- Draw2D——（org.eclipse.draw2d.*）一个基于SWT的轻量级的框架，用于呈现图形信息。
- GEF框架——（org.eclipse.gef.*）一个基于Draw2D的MVC框架，用于处理与图形信息的用户交互。
- Zest——（org.eclipse.zest.*）一个基于Draw2D的图形框架，用于绘制图形。在本书中将不讨论使用Zest。

你可以通过下载并解压缩至Eclipse安装，或通过使用Eclipse更新管理器（为了了解更多信息，参见<http://www.eclipse.org/gef/>并在左侧的导航栏中点击Installation），以安装一个或多个这些GEF子集。在我们的情况中，我们使用Help > Software Updates...命令安装“GEF 3.4 All-In-One SDK”。

20.2 GEF模型

GEF提供了广泛的模型对象，从普通Java对象（plain-old-java-objects, POJO）集到使用Eclipse建模框架（Eclipse Modeling Framework, EMF）开发的复杂模型。然而，基于该原因，在设计与实现模型时有推荐的标准。由于编辑部分和图案是基于模型构建的，功能和能力也由模型限制。首先，模型负责提供所有将由用户可见并更改的数据；在控制器或视图中不应存储任何数据。然后，模型需要提供一种方法让数据在会话间保留。最后，控制器将具有对模型的引用，而模型不应引用控制器或视图。作为替代，控制器将它自己注册为模型的一个监听器，以便于使用观察者样式向控制器传递模型更改。

使用上面列出的准则，有两项内容是收藏夹模型（参见7.2.3节）当前所不具备的：收藏夹模型不具备收藏夹项节点的大小和位置数据；不具备模型更改的通知系统（图20-1）。对于该示例的目的而言，大小和位置数据将由FavoritesManager控制器创建并不会在会话间保留。虽然监听器不能建立以监听所有类型的模型更改，但FavoritesManager确实支持对于收藏夹项的添加和移除的监听：

- addFavoritesManagerListener(FavoritesManagerListener)
- removeFavoritesManagerListener(FavoritesManagerListener)

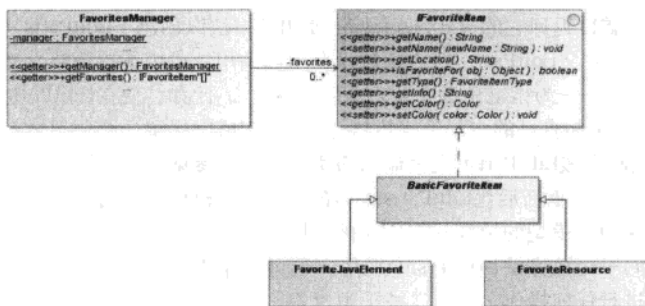


图20-1 收藏夹模型

20.3 GEF控制器

GEF控制器是一个编辑部分（edit part）的集合。当新对象被添加至模型时，编辑部分工厂创建对应于新的模型对象的编辑部分。每个编辑部分负责创建代表它的模型对象的图案，并通过覆盖声明以下内容的方法为它自己设置相关属性：

- 子编辑部分（参见20.3.3节）
- 连接（参见20.3.4节和20.4.4节）
- 图案（参见20.4节）
- 策略（参见20.6.5节）

20.3.1 EditPart类

GEF提供了不同的编辑部分类让用户根据想要的可视化的表示扩展。AbstractGraphicalEditPart根据将呈现图形数据（比如图案）的部分而被继承。而AbstractConnectionEditPart根据展示两个图

案间的连接（参见20.3.4节）而被继承。AbstractEditPart是这些编辑部分的共同超类。

1. AbstractEditPart

activate()——当EditPart成为编辑部分的活动层次结构中的元素时，调用该方法。这些编辑部分的图案将在画布中显示。

- addEditPartListener(EditPartListener listener)——添加一个监听器至EditPart。
- createEditPolicies()——创建初始编辑策略和/或为动态编辑策略保留空间。
- deactivate()——当EditPart不再是编辑部分的活动层次结构的元素时，调用该方法（参见activate）。
- getChildren()——返回子EditParts的List。该方法应仅在内部调用，或由比如编辑处理之类的帮助器调用。
- getCommand(Request request)——返回GEF命令以执行指定的请求或null。
- getModel()——返回该EditPart表示的主模型对象。EditPart可能对应超过一个的模型对象，甚至不对应模型对象。实际上，返回的Object由其他EditParts使用以鉴定该EditPart。此外，编辑策略可能依赖于该方法以构建操作模型的命令。
- getModelChildren()——返回一个包含子模型对象的List。如果该EditPart的模型是一个容器，那么该方法应被覆盖以返回它的后代。refresh()使用该方法以创建子EditParts（参见20.3.2节）。
- getParent()——返回父EditPart。该方法应仅在内部调用，或由编辑策略之类的帮助器调用。
- installEditPolicy()——为指定角色安装一个EditPolicy。比如，使用EditPolicy.LAYOUT_ROLE以在EditPart中安装你自己的布局编辑策略。指定null值以保留一个位置。
- refresh()——刷新由该EditPart可视化显示的所有属性（参见20.5.1节）。
- refreshChildren()——更新后代EditParts的集合以使它与模型后代的集合同步。从refresh()调用，也可以作为对来源于模型的通知的响应而被调用。
- refreshVisuals()——该方法由refresh()调用，也可以作为对来源于模型的通知的响应而调用。该方法默认不进行任何操作，并可以被子类覆盖。
- removeEditPartListener(EditPartListener listener)——从监听器列表移除指定监听器的第一次出现。如果该监听器不存在则不进行任何操作。

2. AbstractGraphicalEditPart

作为AbstractEditPart子类的AbstractGraphicalEditPart，提供了附加方法以处理图案和连接：

- createFigure()——创建Figure以作为该部分的可视部分使用。如果该图案还没有创建，则从getFigure()调用该方法。
- getFigure()——返回Figure以作为该部分的可视部分使用。如果图案当前是null则调用createFigure()。
- getModelSourceConnections()——返回连接模型对象的List。该EditPart的模型是这些连接模型对象的源。refreshSourceConnections()调用该方法。对于每一个连接模型对象，将自动调用createConnection(Object)以获取一个对应的ConnectionEditPart。
- getModelTargetConnections()——返回连接模型对象的List。该EditPart的模型是这些连接模型对象的目标。refreshTargetConnections()调用该方法。对于每个连接模型对象，将自动调用createConnection(Object)以获取一个对应的ConnectionEditPart。

- `setLayoutConstraint(EditPart, IFigure, Object)`——为该`GraphicalEditPart`设置某个后代位于内容页图案的Figure的指定常量。

20.3.2 最高级EditPart

在已经设计好了模型之后，创建编辑部分的集合是相对简单的。根据经验，将在GEF画布上显示的每个模型对象都具有一个对应的编辑部分。对于收藏夹模型而言，有一个最高级编辑部分管理子编辑部分的集合。每个收藏夹项、每个被引用的资源和每个收藏夹项和资源间的连接都有一个子编辑部分（图20-2）。

该基于Favorites视图的GEF的所有代码将被放置于一个独立的项目中，所以我们首先创建一个新的名为“com.qualityeclipse.favorites.gef”的Eclipse插件项目。该项目依赖以下插件：

- org.eclipse.core.resources
- org.eclipse.core.runtime
- org.eclipse.gef
- org.eclipse.jdt.core
- org.eclipse.ui
- org.eclipse.ui.ide
- org.eclipse.ui.views
- com.qualityeclipse.favorites

然后，为我们所有将要扩展的图形编辑部分定义一个抽象类：

```
package com.qualityeclipse.favorites.gef.parts;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public abstract class AbstractFavoritesGraphicalEditPart
    extends AbstractGraphicalEditPart
{
}
```

提示 通过为GEF程序中所有的编辑部分声明一个超类，层次结构视图可以用于在编辑部分、模型和图案间，比使用包管理器视图更简单地导航。

总体上，模型对象的命名约定应在从它们所创建的编辑部分中反映。比如，FavoritesManager的编辑部分应被命名为FavoritesManagerEditPart：

```
public class FavoritesManagerEditPart
    extends AbstractFavoritesGraphicalEditPart
{
    public FavoritesManagerEditPart(FavoritesManager manager) {
        setModel(manager);
    }
    public FavoritesManager getFavoritesManager() {
        return (FavoritesManager) getModel();
    }
    protected IFigure createFigure() {
        // see end of Section 20.4.5, LayoutManager, on page 751
        return null;
    }
    protected void createEditPolicies() {
        // see Section 20.6.5.3, GEF Policies, on page 765
    }
}
```

getModelChildren()

GEF调用getModelChildren()以确定FavoritesManagerEditPart是否具有后代。该方法返回一个模型对象的集合。这些模型对象被传递至编辑部分工厂（参见20.3.5节）以初始化编辑部分。该getModelChildren()方法可以在子编辑部分上实现，以使它们可以返回它们自己的后代，等等。在我们的情况中，FavoritesManagerEditPart将返回一个IFavoriteItems和与那些IFavoriteItems关联的IResources集合。

```
protected List<Object> getModelChildren() {
    IFavoriteItem[] items = getFavoritesManager().getFavorites();
    Collection<Object> result = new HashSet<Object>(items.length * 2);
    for (int i = 0; i < items.length; i++) {
        IFavoriteItem each = items[i];
        result.add(each);
        result.add(each.getAdapter(IResource.class));
    }
    return new ArrayList<Object>(result);
}
```

getModelChildren()方法仅由refreshChildren()方法调用。而refreshChildren()方法又仅由refresh()方法调用。仅当编辑部分首先显示时才调用refresh()方法。

20.3.3 子EditParts

我们想要从收藏夹模型（参见20.2节）和IResource模型对象一起显示FavoriteJavaElement和FavoriteResource。为了完成该任务，我们必须为每一个这些模型对象创建一个编辑部分。

由于这些所有的对象具有类似的参数列表，并将很可能相互连接，我们为这些编辑部分定义一个公共的超类：

```
public abstract class AbstractFavoritesNodeEditPart extends
    AbstractFavoritesGraphicalEditPart
{
}
```

然后，定义BasicFavoriteItemEditPart以用于FavoriteJavaElement和FavoriteResource模型对象。

```
public class BasicFavoriteItemEditPart extends
    AbstractFavoritesNodeEditPart
{
    public BasicFavoriteItemEditPart(BasicFavoriteItem item){
        setModel(item);
    }
    public BasicFavoriteItem getBasicFavoriteItem() {
        return (BasicFavoriteItem) getModel();
    }
    protected IFigure createFigure() {
        // see Section 20.4.3.1, Composing Figures, on page 747
        return null;
    }
    protected void createEditPolicies() {
        // see Section 20.6.9, Deleting model objects, on page 774
    }
}
```

最后，定义与Eclipse资源模型对象IResource对应的ResourceEditPart。

```
public class ResourceEditPart extends
```

```

        AbstractFavoritesNodeEditPart
    {
        public ResourceEditPart(IResource resource) {
            setModel(resource);
        }

        public IResource getResource() {
            return (IResource) getModel();
        }

        protected IFigure createFigure() {
            // see Section 20.4.3.2, Custom Figures, on page 748
            return null;
        }
        protected void createEditPolicies() {
        }
    }

```

20.3.4 连接EditParts

当前，有编辑部分可以代表所有的图案，除了收藏夹部分和资源部分间的连接图案。为了创建连接编辑部分。已有的编辑部分需要使用某些模型对象传递回GEF以通知框架该连接的存在。该通知方式与FavoritesManagerEditPart通知GEF它有后代的方式类似。对于收藏夹模型而言，绘制连接所需的所有信息已经存在于模型中，但没有收藏夹模型对象表示连接。我们不修改模型以包括一个新的模型对象，而是创建一个新的临时对象以代表收藏夹模型对象间的连接。

```

public class FavoritesConnection
{
    private final BasicFavoriteItem source;
    private final IResource target;

    public FavoritesConnection(BasicFavoriteItem item,
        IResource resource) {
        this.source = item;
        this.target = resource;
    }

    public BasicFavoriteItem getBasicFavoriteItem() {
        return source;
    }

    public IResource getResource() {
        return target;
    }
}

```

连接可视化地链接一个“源”和一个“目标”。具有连接的编辑部分必须通过覆盖getModelSourceConnections()和getModelTargetConnections()方法以返回这些连接模型。每个连接模型对象必须由仅一个编辑部分的getModelSourceConnections()方法和仅一个编辑部分的getModelTargetConnections()方法返回。在这里，每个连接的源是BasicFavoriteItemEditPart的一个实例。添加以下字段和方法。

```

private final List<FavoritesConnection> modelSourceConnections;
protected List<FavoritesConnection> getModelSourceConnections() {

```

```

    return modelSourceConnections;
}

```

至BasicFavoriteItemEditPart并在构造函数中初始化该字段。

```

public BasicFavoriteItemEditPart(BasicFavoriteItem item) {
    setModel(item);
    IResource res =(IResource)item.getAdapter(IResource.class);
    modelSourceConnections =new ArrayList<FavoritesConnection>(1);
    modelSourceConnections.add(new FavoritesConnection(item,res));
}

```

BasicFavoriteItemEditPart总是仅具有一个源连接，而ResourceEditPart可以具有多个目标连接。

添加以下字段和方法至ResourceEditPart。字段内容将由这一节稍后的方法所修改。

```

private final List<FavoritesConnection> modelTargetConnections =
    new ArrayList<FavoritesConnection>();

protected List<FavoritesConnection> getModelTargetConnections() {
    return modelTargetConnections;
}

```

现在，创建对应于临时FavoritesConnection对象的连接编辑部分。

```

public class FavoriteConnectionEditPart extends
    AbstractConnectionEditPart
{
    public FavoriteConnectionEditPart(FavoritesConnection connection) {
        setModel(connection);
    }

    protected void createEditPolicies() {
        // none
    }
}

```

当对象被添加至一个已有的模型时，编辑部分工厂（参见20.3.5节）实例化并初始化一个该模型对象新的编辑部分和与该编辑部分相关的连接。当源编辑部分被初始化时，它设置它自己为该连接的源。当初始化目标编辑部分时，它设置它自己为该路径的目标。当激活编辑部分时，所有该编辑部分的源连接都将激活，但目标连接将不会激活，以保证不会两次激活这些连接。

当激活我们的连接编辑部分时，保证设置了连接的源。这是因为连接的模型对象由BasicFavoriteItemEditPart中的getModelSourceConnections返回，但连接的目标还未设置。在FavoriteConnectionEditPart中添加以下方法以设置目标。如果目标还不可用，该方法添加一个监听器以等待该目标可用。

```

public void activate() {
    super.activate();
    final EditPart manager =
        (EditPart) getParent().getChildren().get(0);
    for (Iterator<?> iter = manager.getChildren().iterator();
        iter.hasNext();) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) iter.next();
        if (child.addFavoritesTargetConnection(this))
            return; // target found... no need for listener
    }
}

```

```

manager.addEditPartListener(new EditPartListener.Stub() {
    public void childAdded(EditPart editPart, int index) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) editPart;
        if (child.addFavoritesTargetConnection(
            FavoriteConnectionEditPart.this))
            manager.removeEditPartListener(this);
    }
});
}

```

上面的方法调用了AbstractFavoritesNodeEditPart中的一个新方法。该AbstractFavoritesNodeEditPart在合适的时候添加指定的目标连接，并在连接被添加时返回true。

```

public boolean addFavoritesTargetConnection(
    FavoriteConnectionEditPart editPart) {
    return false;
}

```

然后在ResourceEditPart中覆盖该方法以提供真正的实现。

```

public boolean addFavoritesTargetConnection(
    FavoriteConnectionEditPart editPart) {
    FavoritesConnection conn = editPart.getFavoritesConnection();
    if (!conn.getResource().equals(getResource()))
        return false;
    modelTargetConnections.add(conn);
    addTargetConnection(editPart, 0);
    return true;
}

```

当连接被设置为不可用时，它们的目标编辑部分也必须以一种类似的样式更新。添加以下方法至FavoriteConnectionEditPart:

```

public void deactivate() {
    final EditPart manager =
        (EditPart) getParent().getChildren().get(0);
    for (Iterator<?> iter = manager.getChildren().iterator();
        iter.hasNext();) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) iter.next();
        if (child.removeFavoritesTargetConnection(this))
            break; // target removed... no need to look further
    }
    super.deactivate();
}

```

该方法调用AbstractFavoritesNodeEditPart中的一个新方法:

```

public boolean removeFavoritesTargetConnection(
    FavoriteConnectionEditPart conn) {
    return false;
}

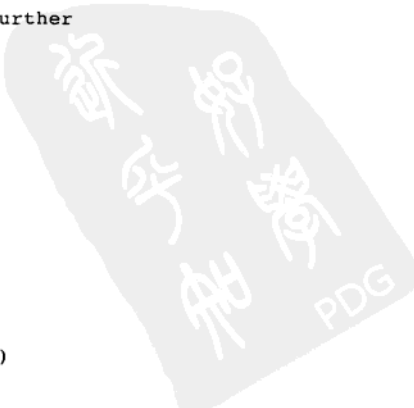
```

在ResourceEditPart中覆盖该方法。

```

public boolean removeFavoritesTargetConnection(
    FavoriteConnectionEditPart conn) {
    if (!modelTargetConnections.remove(conn.getModel()))
        return false;
}

```




```

        removeTargetConnection(conn);
        return true;
    }
}

```

20.3.5 EditPartFactory

每个GEF画布仅有一个编辑部分工厂。当对象被添加至模型时，GEF使用相关的编辑部分工厂创建对应于新模型对象的编辑部分。我们首先声明一个新的编辑部分工厂类：

```

public class FavoritesEditPartFactory
    implements EditPartFactory
{
}

```

在前面的小节，我们为每个模型对象创建了编辑部分以显示于我们的Favorites GEF视图（图20-2）中。在FavoritesEditPartFactory类中，声明一个新的createEditPart()方法为编辑部分所代表的模型对象实例化这些编辑部分。如果没有为某个特定模型对象定义一个编辑部分，将抛出一个IllegalStateException。

```

public EditPart createEditPart(EditPart context, Object model) {

    if (model instanceof FavoritesManager)
        return new FavoritesManagerEditPart((FavoritesManager) model);

    if (model instanceof BasicFavoriteItem)
        return new BasicFavoriteItemEditPart((FavoriteJavaElement) model);

    if (model instanceof IResource)
        return new ResourceEditPart((IResource) model);

    if (model instanceof FavoritesConnection)
        return new FavoriteConnectionEditPart((FavoritesConnection) model);

    throw new IllegalStateException(
        "Couldn't create an edit part for the model object: "
        + model.getClass().getName());
}

```

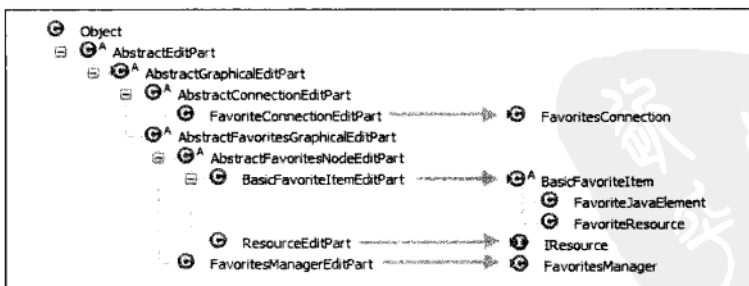


图20-2 具有对应编辑部分的收藏夹模型

20.4 GEF图案

作为FigureCanvas实例的GEF画布，包含一个图案的集合。这些图案视觉上代表底层的模型。

每个编辑部分负责创建和管理代表相关的模型对象的图案。

20.4.1 IFigure

GEF画布中的每个图案必须实现IFigure接口。Graphics类、IFigure接口和许多实际的图案类在GEF的Draw2D部分中定义。由IFigure声明的一些有用的方法包括:

- add(IFigure)——添加给定的IFigure作为该IFigure的后代,与调用add(IFigure, Object, int)类似。
- containsPoint(int, int)——如果point(x, y)包含在该IFigure的边界之内,则返回true。
- getBackgroundColor()——返回背景颜色。
- getBounds()——返回完整包括图案在内的最小矩形。该方法的调用者不能修改返回的Rectangle。
- getClientArea()——返回将在该Figure的边界内放置后代以及后代图案的绘制将被剪切至的矩形区域(通过LayoutManagers)。
- getForegroundColor()——返回前景颜色。
- getParent()——返回包含该IFigure的IFigure。如果没有父部件,返回null。
- getToolTip()——返回作为该IFigure的提示的IFigure。
- hasFocus()——如果该IFigure拥有焦点则返回true。
- isOpaque()——如果该IFigure是不透明的,则返回true。如果该图案是透明的,则返回false。
- isVisible()——如果该图案的可见性标志位被设置为true,则返回true,不递归检查该图案的父部件的可见性。
- paint(Graphics)——绘制该IFigure和它的后代。
- setBackgroundColor(Color)——设置该图案的背景颜色。
- setConstraint(IFigure, Object)——用于设置当前LayoutManager中的指定后代的常量的方便方法。
- setForegroundColor(Color)——设置该图案的前景颜色。
- setLayoutManager(LayoutManager)——设置LayoutManager(参见20.4.5节)。
- setOpaque(boolean)——如果参数是true,则设置该IFigure为不透明的。如果参数是false,则设置为透明的。
- setToolTip(IFigure)——设置一个提示,当鼠标悬停于该IFigure时显示。
- setVisible(boolean)——设置该IFigure的可见性。

20.4.2 Graphics

当呈现图案时,Graphics类提供了许多绘制元素以用于如显示文本、绘制线和绘制并填充多边形之类的任务。一些有用的Graphics方法包括:

- drawFocus(Rectangle)——绘制一个聚焦矩形。
- drawImage(Image, Point)——在一个点绘制给定图像。
- drawLine(Point, Point)——使用当前前景颜色在两个指定点绘制一条线。
- drawPolygon(PointList)——绘制一个由包含顶点的给定PointList定义的封闭多边形。列表中的第一个和最后一个点将连接。
- drawRectangle(Rectangle)——使用当前前景颜色绘制给定的矩形。
- drawRoundRectangle(Rectangle, int, int)——使用当前前景颜色绘制一个圆角矩形。后两个参数表示圆角的水平和垂直直径。

- `drawText(String, Point)`——使用当前字体和前景颜色在指定位置绘制给定字符串。文本的背景将是透明的。还将执行标签扩展和回车处理。
- `fillPolygon(PointList)`——填充一个由包含顶点的给定PointList定义的封闭多边形。列表中的第一个和最后一个点将连接。
- `fillRectangle(Rectangle)`——使用当前背景颜色填充给定矩形。
- `fillRoundRectangle(Rectangle, int, int)`——使用背景颜色填充一个圆角矩形。后两个参数代表圆角的水平和垂直直径。
- `fillText(String, Point)`——使用当前字体和前景颜色在指定位置绘制给定字符串。文本的背景将使用当前背景颜色填充。还将执行标签扩展和回车处理。
- `getBackgroundColor()`——返回背景颜色。
- `getForegroundColor()`——返回前景颜色。
- `getLineStyle()`——返回线的样式。
- `getLineWidth()`——返回线的当前宽度。
- `rotate(float)`——将坐标反时针旋转给定的角度。所有后续的绘制将在旋转后的坐标中执行。当使用一个旋转后的坐标系时，一些功能是不合法的。为了保留对这些功能的访问，有必要调用`restore()`或`pop()`以返回至一个未旋转的状态。
- `setBackgroundColor(Color)`——设置背景颜色。
- `setBackgroundPattern(Pattern)`——设置用于填充类型图形操作的样式。当该样式由Graphics对象引用时，不能被释放。
- `setForegroundColor(Color)`——设置前景颜色。
- `setForegroundPattern(Pattern)`——为绘制和文本操作设置前景样式。该样式当它由Graphics对象引用时，不能被释放。
- `setLineStyle(int)`——设置线样式为参数。该参数必须是SWT.LINE_SOLID、SWT.LINE_DASH、SWT.LINE_DOT、SWT.LINE_DASHDOT或SWT.LINE_DASHDOTDOT常量之一。
- `setLineWidth(int)`——设置线宽度。
- `translate(Point)`——将接收者的坐标移动给定的x和y数量。所有后续绘制将在移动后的坐标系中执行。由它自己使用的整数转换不需要或不开始SWT中的高级图形系统的使用。它被模拟，直至触发了高级图形。

GEF提供了一些基本的图案类。这些图案类可以被扩展或组合以创建更复杂的图案。这些类包括：

- `Button`——一个一般具有边界的图案。该边界将上下移动以作为对被按下的响应。
- `Ellipse`——一个绘制填充它的边界的椭圆的图案。
- `ImageFigure`——包含图像的图案。当显示不具有任何相关文本的图像时，使用该Figure作为标签的替代。
- `Label`——可以显示文本和/或图像的图案（参见20.4.3节）。
- `Panel`——默认是不透明的图案容器。
- `Polygon`——一个与Polyline类似的图案。区别在于该图案是封闭的，且已被填充。
- `Polyline`——作为一系列线段显示的图案。
- `PolylineConnection`——用于可视化连接其他图案的图案。

- `RectangleFigure`——具有直角边角的矩形图案。
- `RoundedRectangle`——具有圆角边角的矩形图案（参见20.4.3节）。
- `TextFlow`——用于在多条线上显示文本串的图案。
- `Thumbnail`——与原图案比例一致，但缩小显示图像的图案。
- `Triangle`——将它自己显示为三角形的图案。

20.4.3 复杂图案

接下来，必须为每个编辑部分创建图案。一种方法是通过组合较简单的图案而不创建任何自定义图案以创建复杂图案。在其他情况中，为每个将在屏幕上呈现的编辑部分类创建一个特殊的图案类是更有效的办法。在该示例中，我们将为收藏夹项编辑部分使用一种方法，为其他的使用 `ResourceEditPart`。

1. 复合图案

图案可以具有子图案，这使得通过组合较简单的图案以创建复杂图案成为现实。编辑部分可以通过使用 `IFigure.add(IFigure)` 方法直接组合图案。此外，编辑部分的图案自动成为该编辑部分的父部件的图案的后代。也就是说，由于 `FavoritesManagerEditPart` 具有包含图案的子编辑部分，GEF 自动声明这些图案为 `FavoritesManagerEditPart` 的图案的后代。

我们打算显示我们的收藏夹项为包含文本和图像的圆角矩形。`Draw2D` 提供了 `RoundedRectangle` 和 `Label` 类。我们可以将这两个类组合为父/子以实现我们的目标。我们首先修改 `BasicFavoriteItemEditPart` 以添加一个新的字段。该字段将被多个方法引用：

```
private final Label label = new Label();
```

修改构造函数以基于收藏夹项的信息初始化标签文本和图像：

```
public BasicFavoriteItemEditPart(BasicFavoriteItem item) {
    setModel(item);
    label.setText(item.getName());
    label.setIcon(item.getType().getImage());
    IResource res = (IResource) item.getAdapter(IResource.class);
    modelSourceConnections = new ArrayList<FavoritesConnection>(1);
    modelSourceConnections.add(new FavoritesConnection(item, res));
}
```

现在，实现 `BasicFavoriteItemEditPart` 中的 `createFigure()` 方法以创建一个圆角矩形，并添加标签为该矩形中央的一个后代。矩形的客户区域必须被竖直居中地插入矩形内的标签：

```
private static final Insets CLIENT_AREA_INSETS =
    new Insets(10, 10, 21, 21);

protected IFigure createFigure() {
    RoundedRectangle figure = new RoundedRectangle() {
        public Rectangle getClientArea(Rectangle rect) {
            Rectangle clientArea = super.getClientArea(rect);
            clientArea.crop(CLIENT_AREA_INSETS);
            return clientArea;
        }
    };
    figure.setSize(150, 40);
    FlowLayout layout = new FlowLayout();
    layout.setMajorAlignment(FlowLayout.ALIGN_CENTER);
```

```

        layout.setMinorAlignment(FlowLayout.ALIGN_CENTER);
        figure.setLayoutManager(layout);
        label.setTextAlignment(PositionConstants.LEFT);
        figure.add(label);
        return figure;
    }

```

2. 自定义图案

我们要显示资源为一个包含资源名称的“具有折叠边角的矩形”。不幸的是，Draw2D没有提供类似的图案，所以我们必须自己编写。为了更好的重用，我们可以创建一个新的、具有与上一节中类似的Label的FoldedCornerRectangle以使用。作为替代，为了简便，我们创建一个新的ResourceFigure类以直观地表示Favorites GEF视图中的资源。

```

public class ResourceFigure extends Figure
{
    private Label label;

    public ResourceFigure() {
        super();
        label = new Label();
        label.setTextAlignment(PositionConstants.LEFT);
        add(label);
        setLayoutManager(new FreeformLayout());
    }

    public void setText(String text) {
        label.setText(text);
    }

    public void setImage(Image icon) {
        label.setIcon(icon);
    }
}

```

该类的paint()方法在屏幕上绘制图案：

```

private static final int FOLDED_CORNER_LENGTH = 12;

protected void paintFigure(Graphics g) {
    super.paintFigure(g);

    Rectangle r = getClientArea();

    // draw the rectangle without the top left corner
    g.drawLine(r.x, r.y,
        r.x + r.width - FOLDED_CORNER_LENGTH - 1, r.y); // top
    g.drawLine(r.x, r.y,
        r.x, r.y + r.height - 1); // left
    g.drawLine(r.x, r.y + r.height - 1,
        r.x + r.width - 1, r.y + r.height - 1); // bottom
    g.drawLine(r.x + r.width - 1, r.y + FOLDED_CORNER_LENGTH - 1,
        r.x + r.width - 1, r.y + r.height - 1); // right

    // draw the label
    setConstraint(label, new Rectangle(r.x + 10, r.y + 10,
        r.width - 21, r.height - 21));
}

```

```

// draw the folded corner
Point topLeftCorner, bottomLeftCorner, bottomRightCorner;
PointList trianglePolygon;
topLeftCorner =
    new Point(r.x + r.width - FOLDED_CORNER_LENGTH - 1, r.y);
bottomLeftCorner =
    new Point(r.x + r.width - FOLDED_CORNER_LENGTH - 1, r.y
        + FOLDED_CORNER_LENGTH);
bottomRightCorner =
    new Point(r.x + r.width - 1, r.y + FOLDED_CORNER_LENGTH);
trianglePolygon = new PointList(3);
trianglePolygon.addPoint(topLeftCorner);
trianglePolygon.addPoint(bottomLeftCorner);
trianglePolygon.addPoint(bottomRightCorner);

g.setBackgroundColor(ColorConstants.lightGray);
g.fillPolygon(trianglePolygon);

g.drawLine(topLeftCorner, bottomLeftCorner);
g.drawLine(bottomLeftCorner, bottomRightCorner);
g.setLineDash(new int[] { 1 });
g.drawLine(bottomRightCorner, topLeftCorner);
}

```

一旦已经实现了ResourceFigure类, 我们必须修改ResourceEditPart以实例化和返回该新的图案。我们首先添加一个新字段至ResourceEditPart, 以使该图案可以由多个方法引用:

```
private final ResourceFigure resourceFigure = new ResourceFigure();
```

修改构造函数以初始化该图案:

```

public ResourceEditPart(IResource resource) {
    setModel(resource);
    resourceFigure.setText(resource.getName());
}

```

最后, 添加一个新方法以返回该图案:

```

protected IFigure createFigure() {
    resourceFigure.setSize(150, 40);
    return resourceFigure;
}

```

3. 提示

图案可以具有提示 (一个IFigure的实例)。该提示当用户悬停于该图案时出现和消失。添加一个新的AbstractFavoritesGraphicalEditPart方法以创建一个包含该编辑部分的类名称的标签。

```

protected Label createToolTipLabel() {
    Label toolTipLabel = new Label();
    String longName = getClass().getName();
    String shortName = longName.substring(longName.lastIndexOf('.') + 1);
    toolTipLabel.setText(shortName);
    return toolTipLabel;
}

```

由该方法创建的Label被每个子类作为提示使用。修改BasicFavoriteItemEditPart中的createFigure方法以设置该图案的提示。

```
protected IFigure createFigure() {
```

```

    ... existing method statements here ...
    figure.setToolTipText(createToolTipLabel());
    return figure;
}

```

以类似的方法修改ResourceEditPart中的createFigure()。

```

protected IFigure createFigure() {
    resourceFigure.setSize(150, 40);
    resourceFigure.setToolTipText(createToolTipLabel());
    return resourceFigure;
}

```

20.4.4 连接图案

默认地，连接被显示为不具有任何箭头的窄的实心线段。你可以更改该线段的宽度、样式、填充和终点装饰。为了将连接显示为从收藏夹项指向资源的箭头，添加以下方法至FavoriteConnectionEditPart：

```

protected IFigure createFigure() {
    PolylineConnection connection =
        (PolylineConnection) super.createFigure();
    connection.setTargetDecoration(new PolygonDecoration());
    return connection;
}

```

20.4.5 LayoutManager

所有具有后代的图案必须声明一个LayoutManager。该LayoutManager负责放置并调整子图案的大小。GEF提供了许多通用布局管理器，比如：

- BorderLayout
- FreeformLayout——用于20.4.3节
- FlowLayout——用于20.4.3节
- GridLayout
- StackLayout

在我们的Favorites GEF视图中，我们需要资源出现在一列，而收藏夹项出现于右边的另一列（图20-3）。不幸的是，GEF没有提供一个适合于我们情况的布局管理器，所以我们必须创建定制布局管理器。我们从创建一个子类AbstractLayout开始。

```

public class FavoritesLayout extends AbstractLayout
{
}

```

我们的目标是安排图案以使资源在左侧按照字母排序，而相关的收藏夹项排列于右侧，以防止连接相互重叠。分类意味着从模型获取信息，但图案一开始不具有模型的任何信息。该由布局管理器所需要的，但在图案中不可用的附加信息，被称为布局常量，一般通过LayoutManager setConstraint()方法提供。一般地，基于重用的目的，布局管理器不具有使用它的编辑部分的信息，而且我们在该示例中遵循该原则。添加以下FavoritesLayout字段和方法以基于用于分类图案的常量跟踪String。

```

private final Map<IFigure, String> constraints =
    new HashMap<IFigure, String>(20);

```

无论何时添加图案，我们必须设置与该图案关联的常量。添加以下方法至Favorites Manager-EditPart为每个新图案设置常量。

```
protected void addChild(EditPart child, int index) {
    super.addChild(child, index);
    AbstractFavoritesNodeEditPart childEditPart =
        (AbstractFavoritesNodeEditPart) child;
    IFigure childFigure = childEditPart.getFigure();
    LayoutManager layout = getFigure().getLayoutManager();
    String constraint = child.getSortKey();
    layout.setConstraint(childFigure, constraint);
}
```

上面的方法基于getSortKey()方法设置常量。而我们必须实现该方法。添加以下抽象方法至AbstractFavoritesNodeEditPart。

```
public abstract String getSortKey();
```

该方法必须在ResourceEditPart中实现：

```
public String getSortKey() {
    return getResource().getName();
}
```

以及在BasicFavoriteItemEditPart中实现。

```
public String getSortKey() {
    BasicFavoriteItem elem = getBasicFavoriteItem();
    IResource res = (IResource) elem.getAdapter(IResource.class);
    return res.getName() + "," + elem.getName()
        + "," + elem.getType().getId();
}
```

20.5 Eclipse视图中的GEF

一旦定义了GEF模型、编辑部分、编辑部分工厂和图案，我们就可以在Eclipse视图中图形显示收藏夹项（图20-3）。在最高级，GEF提供了Scrolling GraphicalViewer类。它属于编辑部分。它可以用于创建和管理GEF画布。使用ScrollingGraphicalViewer.createControl()方法，画布可以被添加至任意SWT复合组件（参见4.2.6节）。

图形数据的根编辑部分是ScalableRootEditPart或ScalableFreeformRootEditPart。它们都提供了同样的功能以绘制、滚动和缩放。但自由变形图案允许图案具有负坐标，并且是最常用的根编辑部分。这些编辑部分管理GEF画布内的一系列层。每个层具有不同的目标。一个层包含所有的收藏夹图案，而另一个层包含代表收藏夹图案间的连接的图案。

声明一个具有以下属性的新视图（参见7.1.2节）：

category——“com.qualityeclipse.favorites”

class——“com.qualityeclipse.favorites.gef.views.FavoritesGEFView”

id——“com.qualityeclipse.favorites.gef.view”

name——“Favorites GEF View”

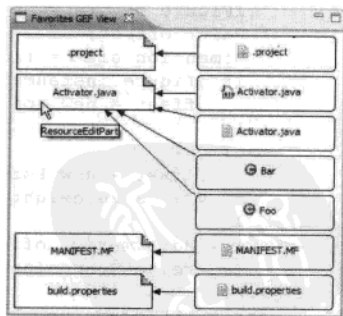


图20-3 显示提示的收藏夹GEF视图

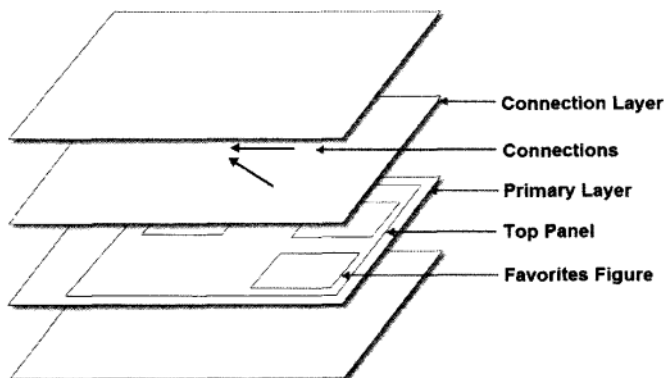


图20-4 层、图案和连接

创建包含以下字段和方法的FavoritesGEFView类（为了了解更多关于视图部分的内容，参见7.2节）。该方法实例化并初始化GEF查看器。

```
ScrollingGraphicalViewer graphicalViewer;

public void createPartControl(Composite parent) {
    ScalableFreeformRootEditPart rootEditPart =
        new ScalableFreeformRootEditPart();
    graphicalViewer = new ScrollingGraphicalViewer();
    graphicalViewer.createControl(parent);
    graphicalViewer.getControl().setBackground(
        ColorConstants.listBackground);
    graphicalViewer.setRootEditPart(rootEditPart);
    graphicalViewer.setEditPartFactory(
        new FavoritesEditPartFactory());
    graphicalViewer.setContents(FavoritesManager.getManager());
}
```

我们的GEF视图包含连接，并且这些连接显示于收藏夹图案之上的层中（图20-4）。默认地，连接是从“源”图案的边缘至“目标”图案边缘的直线。在我们的情况中，这意味着一些连接将显示于收藏夹或资源图案的上方。为了减少这种情况的发生，更改路径路由为ShortestPathConnectionRouter，以使连接将绕过已有图案。添加以下语句至上面定义的createPartControl方法。

```
FavoritesManagerEditPart managerPart = (FavoritesManagerEditPart)
    rootEditPart.getChildren().get(0);
ConnectionLayer connectionLayer = (ConnectionLayer)
    rootEditPart.getLayer(LayerConstants.CONNECTION_LAYER);
connectionLayer.setConnectionRouter(
    new ShortestPathConnectionRouter(managerPart.getFigure()));
```

当该视图第一次可见时，编辑部分和图案被实例化。一般地，当模型更改时，监听器将更改通知编辑部分（参见20.5.1节）。然后编辑部分更新图案，以使图形表示与模型保持同步。现在，我们添加一个setFocus方法至FavoritesGEFView，以使编辑部分和图案在每一次Favorites GEF视图获得焦点时都被刷新。当实现了监听器时，为在该方法中的setContents()方法调用添加注释。

```
public void setFocus() {
```

```
graphicalViewer.setContents(FavoritesManager.getManager());  
}
```

提示 如果一个子图案不是在GEF画布中绘制的,那么请检查:

- 1) 父模型对象是否包含预期的子模型对象。
- 2) 是否返回预期的子模型对象 (参见20.3.2节)。
- 3) 编辑部分过程是否将该子模型对象转换为正确的编辑部分 (参见20.3.5节)。
- 4) 父图案是否具有布局管理器 (参见20.4.5节)。
- 5) 子图案的常量或边界是否被正确设置。

监听模型更改

当模型更改时, 我们想要视图自动调整, 以使它总是显示当前的模型状态。为了完成该任务, 在FavoritesManagerEditPart中创建一个新的监听器以基于模型更改刷新编辑部分。

```
private final FavoritesManagerListener modelListener =  
    new FavoritesManagerListener() {  
        public void favoritesChanged(FavoritesManagerEvent event) {  
            refresh();  
        }  
    };
```

当发生模型更改时, 编辑部分应通过刷新所有来源于模型的更改以实现正确行为。出于简便考虑, 上面描述的模型监听器使用refresh()方法。该方法适用于中小型模型。随着模型不断变大, 执行refresh()方法的次数呈现线性增长。如果某些时候性能变得缓慢, 可以考虑明确调用addChild()和removeChild(), 而不是调用refresh()。

编辑部分具有方法activate()和deactivate()。它们在编辑部分添加和移除时调用。这些方法提供了理想的位置以添加和移除模型监听器。添加以下方法至FavoritesManagerEditPart, 以使它在模型更改时能收到通知。

```
public void activate() {  
    super.activate();  
    getFavoritesManager().addFavoritesManagerListener(modelListener);  
}  
  
public void deactivate() {  
    getFavoritesManager().removeFavoritesManagerListener(modelListener);  
    super.deactivate();  
}
```

我们模型中的资源和收藏夹项是不可改变的, 所以没有必要添加附加监听器。如果资源和收藏夹项模型对象确实发生了更改, 我们将需要添加类似的方法至对应的编辑部分, 以使模型更改能总是在视图中得到反映。

20.6 Eclipse编辑器中的GEF

虽然显示图形信息是很好的, 但如果能操作该图形信息那将是更好的。借助一些额外的工作, GEF向用户提供了与GEF画布中的图案交互的能力。在这一节, 我们从Favorites GEF视图转到Favorites GEF编辑器 (图20-5)。

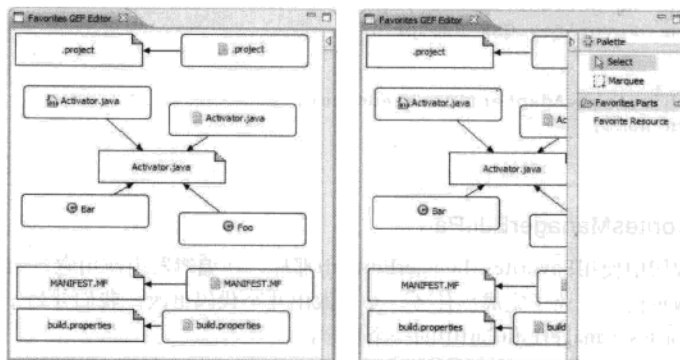


图20-5 具有弹出式选项板的收藏夹GEF编辑器

20.6.1 编辑器输入

根据8.4节，每个编辑器使用一个编辑器输入进行初始化。GEF编辑器也同样如此。编辑器输入提供了模型。在我们的示例中，它是一个FavoritesManager的实例。创建包含我们模型的IEditorInput的一个新子类。

编辑器一般是用于修改某些文件类型，但我们的模型不是存储于一个工作区文件，也不出现于Package Explorer视图中。在我们的编辑器输入类中，exists()方法返回false表示输入不是来源于工作区文件。而getPersistable()方法返回null表示模型没有在工作区文件中保留。

```
public class FavoritesGEFEditorInput
    implements IEditorInput
{
    private final FavoritesManager model;

    public FavoritesGEFEditorInput(FavoritesManager model) {
        this.model = model;
    }

    public FavoritesManager getModel() {
        return model;
    }

    public boolean exists() {
        return false;
    }

    public ImageDescriptor getImageDescriptor() {
        return null;
    }

    public String getName() {
        return "Favorites GEF Editor";
    }

    public IPersistableElement getPersistable() {
        return null;
    }
}
```



```

    public String getToolTipText() {
        return "Favorites GEF Editor";
    }

    public Object getAdapter(Class adapter) {
        return null;
    }
}

```

20.6.2 回到FavoritesManagerEditPart

我们如同在视图中使用FavoritesManagerEditPart那样，在编辑器中使用它。出于该示例的目的，我们想要区分这两种使用。为了完成该任务，必须做出几个代码更改。我们开始于添加editable为构造函数参数和FavoritesManagerEditPart中的一个新字段。

```

private final boolean editable;

public FavoritesManagerEditPart(FavoritesManager manager,
    boolean editable ) {
    setModel(manager);
    this.editable =editable;
}

```

该更改需要修改FavoritesEditPartFactory以具有一个editable构造函数参数和字段。

```

private final boolean editable;

public FavoritesEditPartFactory(boolean editable){
    this.editable =editable;
}

public EditPart createEditPart(EditPart context, Object model) {
    if (model instanceof FavoritesManager)
        return new FavoritesManagerEditPart((FavoritesManager) model,
            editable );
    // the rest of the method is unchanged
}

```

最后，必须修改FavoritesGEFView中的createPartControl方法中的setEditPartFactory语句以传递false。

```

graphicalViewer.setEditPartFactory(new FavoritesEditPartFactory(false));

```

20.6.3 绘画编辑器类

GEF提供了三个类。可以基于这三个类创建编辑器。GraphicalEditor是GraphicalEditorWithPalette和GraphicalEditorWithFlyoutPalette的超类。这两个类，如同它们名字所表达的那样，实现了附加方法以在编辑器中放置选项板。如果在你的编辑器中不需要选项板，那么扩展GraphicalEditor而不是它的一个子类。

1. GraphicalEditor

由该超类提供一些有趣的方法包括：

- createActions()——为该编辑器创建操作。子类应覆盖该方法以使用ActionRegistry创建并注册操作。
- getCommandStack()——返回命令栈。

- `getEditDomain()`——返回编辑域。
- `initializeGraphicalViewer()`——覆盖以在`GraphicalViewer`创建后设置它的内容。
- `setEditDomain(DefaultEditDomain)`——为该`EditorPart`设置`EditDomain`。

2. `GraphicalEditorWithPalette`

该类扩展了`GraphicalEditor`以提供一个固定的选项板。

- `getPaletteRoot()`——必须由子类提供为选项板查看器返回`PaletteRoot`。

3. `GraphicalEditorWithFlyoutPalette`

与`GraphicalEditorWithPalette`类似，该类扩展了`GraphicalEditor`以提供一个选项板。但该选项板可以由用户移动或折叠。

- `getPaletteRoot()`——必须由子类实现，为选项板查看器返回`PaletteRoot`。
- `getPalettePreferences()`——默认地，该方法返回一个`FlyoutPreferences`对象以存储GEF插件中的弹出式图标的位置。子类可以覆盖该方法。

4. 编辑域

GEF编辑器不仅图形显示信息，还使得用户对该信息的操作变得容易。为了帮助编辑器对模型的更改，每个GEF编辑器必须具有一个编辑域。该编辑域提供了接口用于所有用户操作（参见20.6.5节）、命令栈的跟踪（参见20.6.6节）、活动工具和选项板（参见20.7节）。为了我们的目的，由GEF提供的`DefaultEditDomain`类满足我们的所有要求。我们在编辑器的构造函数中实例化该域。

20.6.4 `FavoritesGEFEditor`

我们的编辑器（图20-5）扩展了`GraphicalEditorWithFlyoutPalette`，以使它可以包括一个选项板。操作可以从该选项板中初始化。我们首先创建一个新的编辑器扩展项（参见8.1节）：

- `class`——“`com.qualityeclipse.favorites.gef.editors.FavoritesGEFEditor`”
- `contributorClass`——“`com.qualityeclipse.favorites.gef.editors.FavoritesGEFEditorContributor`”（参见20.6.6节）
- `id`——“`com.qualityeclipse.favorites.gef.editor`”
- `name`——“Favorites GEF Editor”

然后实现`FavoritesGEFEditor`。该编辑器包含一系列与Favorites GEF View类似的语句。Favorites GEF View中的这些语句设置连接路由至一个`ShortestPathConnectionRouter`的实例。

```
public class FavoritesGEFEditor
    extends GraphicalEditorWithFlyoutPalette
{
    public static final String ID =
        "com.qualityeclipse.favorites.gef.editor";

    private FavoritesManager model;

    public FavoritesGEFEditor() {
        setEditDomain(new DefaultEditDomain(this));
    }
    protected void setInput(IEditorInput input) {
        super.setInput(input);
        model = ((FavoritesGEFEditorInput) input).getModel();
    }
}
```

```

protected void configureGraphicalViewer() {
    super.configureGraphicalViewer();
    GraphicalViewer viewer = getGraphicalViewer();
    viewer.setEditPartFactory(new FavoritesEditPartFactory(true));
    viewer.setRootEditPart(new ScalableFreeformRootEditPart());
}

protected void initializeGraphicalViewer() {
    super.initializeGraphicalViewer();

    GraphicalViewer viewer = getGraphicalViewer();
    viewer.setContents(model);

    ScalableFreeformRootEditPart rootEditPart =
        (ScalableFreeformRootEditPart) viewer.getRootEditPart();
    FavoritesManagerEditPart managerPart =
        (FavoritesManagerEditPart) rootEditPart.getChildren().get(0);
    ConnectionLayer connectionLayer = (ConnectionLayer)
        rootEditPart.getLayer(LayerConstants.CONNECTION_LAYER);
    connectionLayer.setConnectionRouter(
        new ShortestPathConnectionRouter(managerPart.getFigure()));
}

public Object getAdapter(Class type) {
    if (type == FavoritesManager.class)
        return model;
    return super.getAdapter(type);
}
}

```

和稍早提到的一样，我们的编辑器不存储内容于工作区文件中。为了阻止程序提示用户保存编辑器内容至工作区文件，添加以下方法。

```

public boolean isDirty() {
    return false;
}

public void doSave(IProgressMonitor monitor) {
    // do nothing
}

```

打开编辑器

收藏夹模型不是存储于工作区文件中，并且我们不能从Package Explorer打开它。因此我们创建一个Open Favorites Editor命令（参见6.1.1节）。

- id——“com.qualityeclipse.favorites.gef.commands.openEditor”
- name——“Open Favorites Editor”
- description——“Open the Favorites editor if it is not already visible”
- categoryId——“com.qualityeclipse.favorites.commands.category”
- defaultHandler——“com.qualityeclipse.favorites.gef.handlers.OpenFavoritesEditorHandler”

以下内容用于创建一个新的菜单添加项，以使上面的命令可以在Favorites菜单中可见（参见6.2.1节）。

- locationURI——“menu:com.qualityeclipse.favorites.menus.favoritesMenu?after=additions”

- `commandId`——“`com.qualityeclipse.favorites.gef.commands.openEditor`”

上面定义的命令具有一个与它相关的默认处理器。如果Favorites GEF Editor还没有打开，那么该处理器将它打开。

```
public class OpenFavoritesEditorHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        try {
            IEditorInput editorInput =
                new FavoritesGEFEditorInput(FavoritesManager.getManager());
            Activator.getDefault()
                .getWorkbench()
                .getActiveWorkbenchWindow()
                .getActivePage()
                .openEditor(editorInput, FavoritesGEFEditor.ID);
        }
        catch (PartInitException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

20.6.5 用户与GEF的交互

当用户操作图案时，GEF系统分解用户交互的连续流为分离的GEF请求。这些请求将分发至由编辑部分注册的合适的GEF策略。当前正操作这些编辑部分的图案。策略处理请求并产生GEF命令。每个命令代表一个底层模型的特定更改。通过将用户交互分解为GEF命令，GEF系统可以保持一个命令栈并减轻Eclipse通用样式的撤销和重做操作。

请注意 GEF命令与第6章中讨论的Eclipse命令框架没有共同之处，唯一相同之处在于它们都出现于Eclipse撤销/重做栈中之外。

1. GEF请求

GEF请求封装关于特定用户交互（如选择、拖动和连接图案）的细节。请求由GEF系统用于获取交互并将该信息传递至合适的GEF策略。正操作的用户交互的类型（角色）和图案用于确定哪一个GEF策略应接收该请求。

2. GEF角色

用户与图案的交互是以几种被称为角色的预定义方式之一进行的。GEF定义了一些角色，如LAYOUT_ROLE，它表示用户如何移动和调整图案的大小；和COMPONENT_ROLE，它表示如何删除图案。每个编辑部分可以为这些角色中之一注册一个策略。这些角色提供了特定操作在用户与该编辑部分的图案交互时执行。预定义GEF角色集包括：

- `EditPolicy.COMPONENT_ROLE`——用于注册定义“基础”操作的策略的键。该操作修改或删除模型元素。一般地，与该角色相关的策略仅了解关于模型的内容，并在该模型上执行操作。图案更改不直接由该策略生成，而是作为监听和响应模型更改时间的编辑部分的结果。我们继承ComponentEditPolicy（参见20.6.9节）以创建一个策略用于删除收藏夹项。
- `EditPolicy.CONNECTION_ROLE`——用于注册定义诸如创建连接的操作的策略的键。

GraphicalNodeEditPolicy提供许多底层功能以用于实现你自己的连接策略。

- EditPolicy.LAYOUT_ROLE——用于注册定义编辑部分的图案上的操作的策略的键。这些操作包括创建、移动和调整大小。我们继承XYLayoutEditPolicy（参见20.6.5节）以创建一个策略用于在GEF编辑器周围移动图案。

3. GEF策略

如果编辑部分需要用户与该编辑部分的图案进行交互，那么该编辑部分应实现createEditPolicies()方法以使用installEditPolicy()方法注册一个或多个GEF策略。在我们的编辑器中，我们想要在屏幕上调整图案大小和重新放置图案的功能。为了实现该功能，实现FavoritesManager-EditPart中的createEditPolicies()方法以注册一个LAYOUT_ROLE。当移动或调整FavoritesManager-EditPart的大小时，GEF系统将请求分发至与该LAYOUT_ROLE关联的策略。

```
protected void createEditPolicies() {  
    installEditPolicy(EditPolicy.LAYOUT_ROLE,  
        new FavoritesLayoutEditPolicy());  
}
```

接下来，创建FavoritesLayoutEditPolicy类以接收并处理这些请求。我们继承XYLayoutEditPolicy以提供大部分的图案操作行为。我们的子类转换常量更改（图案的大小或位置的更改）为更新底层收藏夹模型的更新。更具体地说，GEF系统分发ChangeBoundsRequests至我们的策略。XYLayoutEditPolicy超类处理每个ChangeBoundsRequest并分发编辑部分和它的新常量至createChangeConstraintCommand()方法。我们子类的createChangeConstraintCommand()方法将该信息转换为一个新的AdjustConstraintCommand或在不可以执行操作时返回UnexecutableCommand.INSTANCE。

```
public class FavoritesLayoutEditPolicy extends XYLayoutEditPolicy  
{  
    protected Command createChangeConstraintCommand(  
        EditPart child, Object constraint) {  
        if (child instanceof AbstractFavoritesNodeEditPart) {  
            if (constraint instanceof Rectangle) {  
                return new AdjustConstraintCommand(  
                    (AbstractFavoritesNodeEditPart) child,  
                    (Rectangle) constraint);  
            }  
        }  
        protected Command getCreateCommand(CreateRequest request) {  
            // revisited in Section 20.7.2, CreateCommand, on page 776  
            return UnexecutableCommand.INSTANCE;  
        }  
    }  
}
```

4. GEF命令

GEF命令封装可以应用或撤销的模型更改。如同org.eclipse.gef.commands.Command的子类，每个命令具有一些它可以覆盖以提供必要行为的方法。

- canExecute()——如果命令可以被执行，那么值为真。
- canUndo()——如果命令可以被撤销，那么值为真。该方法应仅在调用execute()或redo()之后调用。
- chain(Command)——返回一个代表一个指定Command至该Command的链的Command。被链接的Command将在该命令执行后execute()，在该Command被撤销之前将undo()。

- execute()——执行Command。该方法在Command是不可执行时不应被调用。
- redo()——重新执行该Command。该方法应仅在已经调用了undo()之后调用。
- setLabel()——设置用于向用户描述该命令的标签。
- undo()——撤销在execute()中执行的更改。该方法应仅在已经调用执行之后，并且canUndo()返回true时调用。

每个GEF命令代表一个底层模型的指定更改。GEF命令应仅引用和修改模型，而不是图案或编辑部分。图案的更改应仅作为编辑部分响应模型更改事件的结果。也就是说，我们的模型不存储图案的大小或位置，因此AdjustConstraintCommand必须直接引用与修改编辑部分。

```
public class AdjustConstraintCommand extends Command
{
    private GraphicalEditPart editPart;
    private Rectangle newBounds, oldBounds;

    public AdjustConstraintCommand(
        GraphicalEditPart editPart, Rectangle constraint) {
        this.editPart = editPart;
        this.newBounds = constraint;
        this.oldBounds = new Rectangle(editPart.getFigure().getBounds());
        setLabel(getOp(oldBounds, newBounds) + getName(editPart));
    }
    private String getOp(Rectangle oldBounds, Rectangle newBounds) {
        if (oldBounds.getSize().equals(newBounds.getSize()))
            return "Move";
        return "Resize ";
    }
    private static String getName(EditPart editPart) {
        Object model = editPart.getModel();
        if (model instanceof IFavoriteItem)
            return ((IFavoriteItem) model).getName();
        if (model instanceof IResource)
            return ((IResource) model).getName();
        return "Favorites Element";
    }
}

public void execute() {
    redo();
}

public void redo() {
    ((GraphicalEditPart) editPart.getParent()).setLayoutConstraint(
        editPart, editPart.getFigure(), newBounds);
}

public void undo() {
    ((GraphicalEditPart) editPart.getParent()).setLayoutConstraint(
        editPart, editPart.getFigure(), oldBounds);
}
}
```

上面的命令假设特定模型的编辑部分不随时间而改变。这是一个无充分根据的假设，因为如果一个模型对象从模型中移除，那么引用该模型的编辑部分将被释放。如果移除该模型对象的操作被撤销，它将重新添加该同一个模型对象至该模型，但一个新的编辑部分将被关联至该同样的模型对

象。如果发生了这种情况，那么上面的操作将不再是可撤销的。解决该问题的一种方法是缓存该模型对象和FavoritesManagerEditPart（它在编辑器的生命周期中保持恒定），而不是编辑部分，然后在每一次它需要的时候查找编辑部分。

```
public class AdjustConstraintCommand extends Command
{
    private FavoritesManagerEditPart manager;
    private Object model;
    private Rectangle newBounds, oldBounds;

    public AdjustConstraintCommand(
        GraphicalEditPart editPart, Rectangle constraint) {
        this.manager =(FavoritesManagerEditPart)editPart.getParent();
        this.model =editPart.getModel();
        this.newBounds = constraint;
        this.oldBounds = new Rectangle(editPart.getFigure().getBounds());
        setLabel(getOp(oldBounds, newBounds) + " " + getName(editPart));
    }

    ... getOp(), getName() and execute() here from prior code listing ...
    public void redo() {
        GraphicalEditPart editPart =getEditPart();
        if (editPart ==null)
            return;
        manager .setLayoutConstraint(
            editPart, editPart.getFigure(), newBounds);
    }
    public void undo() {
        GraphicalEditPart editPart =getEditPart();
        if (editPart ==null)
            return;
        manager .setLayoutConstraint(
            editPart, editPart.getFigure(), oldBounds);
    }

    private GraphicalEditPart getEditPart(){
        for (Iterator<?>iter =manager.getChildren().iterator();
            iter.hasNext());{
            GraphicalEditPart editPart =(GraphicalEditPart)iter.next();
            if (model.equals(editPart.getModel()))
                return editPart;
        }
        return null;
    }
}
```

而另一种方法将把位置和大小信息推送至模型。如果是这种情况，基于该信息将通过模型更改事件流至编辑部分，我们可以修改该模型并忽略编辑部分。如果不能修改底层模型，那么考虑存储该信息于一个基于底层模型之上的新的模型中。

20.6.6 编辑菜单

GEF使用GEF命令跟踪模型更改。但对于撤销命令，我们必须关联GEF至Eclipse编辑框架（图20-6）。关联撤销/重做命令需要一个编辑器添加者，而关联删除目录可以使用新的命令基础结构完成。

1. 撤销和重做命令

GEF提供ActionBarContributor类以较易地关联至Edit > Undo和Edit > Redo命令。我们在20.6.4节中声明的编辑器添加者，扩展了ActionBarContributor类。

```
public class FavoritesGEFEditorContributor extends ActionBarContributor
{
    protected void buildActions() {
    }
    protected void declareGlobalActionKeys() {
        addGlobalActionKey(ActionFactory.UNDO.getId());
        addGlobalActionKey(ActionFactory.REDO.getId());
    }
}
```

2. 删除命令

我们还想要Edit > Delete命令以删除我们编辑器中的选中图案。我们可以使用上面描述的FavoritesGEFEditorContributor以关联全局删除目录。但在这里，我们使用新的命令基础结构作为替代。添加以下命令处理器声明至插件清单。该插件清单将Edit > Delete命令关联至一个新的处理器（参见6.3节以了解更多关于命令处理器的内容，并参见8.5.3节以了解类似示例）。activeWhen表达式指定处理器应仅在我们的编辑器是活动编辑器时才可用。

```
<handler class=
    "com.qualityeclipse.favorites.gef.handlers.FavoritesGEFDeleteHandler"
    commandId="org.eclipse.ui.edit.delete">
    <activeWhen>
        <with
            variable="activeEditorId">
                <equals
                    value="com.qualityeclipse.favorites.gef.editor">
                </equals>
            </with>
        </activeWhen>
    </handler>
```

上面的声明引用了一个新的处理器：

```
public class FavoritesGEFDeleteHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        IEditorPart editor = HandlerUtil.getActiveEditor(event);
        if (editor instanceof FavoritesGEFEditor)
            ((FavoritesGEFEditor) editor).deleteSelection();
        return null;
    }
}
```

该方法调用了FavoritesGEFEditor中的一个新方法。

```
public void deleteSelection() {
    getActionRegistry().getAction>DeleteAction.ID).run();
}
```

一旦我们关联底层策略和20.6.9节中的命令，该删除操作将变得有用。

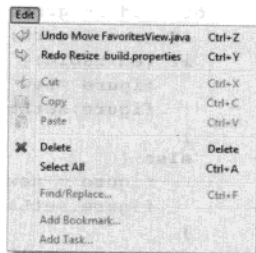


图20-6 具有GEF撤销/重做命令的编辑菜单

20.6.7 FreeformLayer和FreeformLayout

FavoritesLayout将图案放置于特殊位置，并且不允许这些图案被移动。实际上，FavoritesLayout与上面的命令是不协调的，并且如果移动或调整图案的大小时，则以下异常将出现于日志中。

```
java.lang.ClassCastException:
    com.qualityeclipse.favorites.gef.layouts.FavoritesLayout
    cannot be cast to org.eclipse.draw2d.XYLayout
```

对于我们的编辑器，我们需要一个允许重新放置图案的布局管理器。在Favorites Manager-EditPart中，替换图案和它的布局管理器分别为FreeformLayer和FreeformLayout。

```
protected IFigure createFigure() {
    IFigure figure;
    if (editable){
        figure =new FreeformLayer();
        figure.setLayoutManager(new FreeformLayout());
    }
    else {
        figure = new Panel();
        figure.setLayoutManager(new FavoritesLayout());
    }
    figure.setToolTip(createToolTipLabel());
    return figure;
}
```

FreeformLayout类要求是Rectangles的常量为每个图案指定边界区域。而addChild()的当前实现设置适合于FavoritesLayout的常量。做出下列修改以防止在addChild()方法中设置常量。

```
protected void addChild(EditPart child, int index) {
    super.addChild(child, index);
    if (editable)
        return;
    AbstractFavoritesNodeEditPart childEditPart =
        (AbstractFavoritesNodeEditPart) child;
    IFigure childFigure = childEditPart.getFigure();
    LayoutManager layout = getFigure().getLayoutManager();
    String constraint = child.getSortKey();
    layout.setConstraint(childFigure, constraint);
}
```

现在图案可以移动或调整大小。但一开始它们都位于编辑器的左上角并相互重叠。我们需要一个由FavoritesLayout提供的初始布局。添加一个从activate()方法中调用的新方法。该方法使用FavoritesLayout执行一个初始布局，然后为FreeformLayout设置常量。

```
public void activate() {
    super.activate();
    if (editable)
        cleanupLayout();
    getFavoritesManager().addFavoritesManagerListener(modelListener);
}

private void cleanupLayout() {
    LayoutManager layout = new FavoritesLayout();
    for (Iterator<?> iter = getChildren().iterator(); iter.hasNext();) {
        AbstractFavoritesNodeEditPart child =
```

```

        (AbstractFavoritesNodeEditPart) iter.next();
        IFigure childFigure = child.getFigure();
        String constraint = child.getSortKey();
        layout.setConstraint(childFigure, constraint);
    }
    layout.layout(getFigure());

    layout = getFigure().getLayoutManager();
    for (Iterator<?> iter = getChildren().iterator(); iter.hasNext();) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) iter.next();
        IFigure childFigure = child.getFigure();
        Rectangle constraint = childFigure.getBounds();
        layout.setConstraint(childFigure, constraint);
    }
}

```

20.6.8 z顺序

在GEF中，x轴是从左向右延伸（当你向右移动时x坐标的值将增加），而y轴是从上至下的（当你向下移动时y坐标的值将增加），原点（0,0）一般位于GEF画布的左上角。GEF中没有z轴，但在概念上，z轴将从你的显示器以正确的角度向x轴和y轴延伸。由于GEF图案可以相互重叠。存在一个概念上的图案的z顺序以确定哪个图案在另一个的上面。

GEF中的每个编辑部分可以具有包含零个或多个子编辑部分的列表。列表中的第一个或较早的后代被认为具有一个较低的z顺序，并因此绘制于列表中较后的那些后代的下面。当调用refreshChildren()方法时，要么父编辑部分成为活动的，要么精确调用refresh()，重新排序子编辑部分的列表以匹配由getModelChildren()返回的模型对象的顺序。

当添加模型对象后，一般的用户想要代表这些模型对象的图案出现于已有对象的上面。FavoritesManagerEditPart中的getModelChildren()方法由HashSet支持，因此由该方法返回的模型对象的顺序是不确定的。当添加新对象至模型时，我们的模型监听器调用refresh()，该方法随后调用getModelChildren()，因此代表那些新模型对象的图案以一种随机z顺序出现，而不是位于上面。此外，这些新图案的常量还没有设置，所以它们重叠于编辑器的左上角。

一种解决刚刚描述的z顺序和位置问题的方法是使用addChild()和removeChild()而不是调用refresh()来更新子编辑部分。在FavoritesManagerEditPart中，修改已有的模型监听器并添加一个新的updateChildren()方法。该新方法必须移除对应于移除的收藏夹项的编辑部分，并添加对应于添加的收藏夹项的新编辑部分。此外，如果一个资源编辑部分不再具有任意引用它的收藏夹编辑部分，那么它也必须移除。此外，新图案将被放置于随机位置，而不是重叠于左上角。

```

private final FavoritesManagerListener modelListener =
    new FavoritesManagerListener() {
        public void favoritesChanged(FavoritesManagerEvent event) {
            if (editable)
                updateChildren(event);
            else
                refresh();
        }
    };

private void updateChildren(FavoritesManagerEvent event) {

```

```

// Build a map of model object to child edit parts
Map<Object, EditPart> modelToEditPart =
    new HashMap<Object, EditPart>(getChildren().size());
for (Iterator<?> iter = getChildren().iterator(); iter.hasNext();)
{
    GraphicalEditPart child = (GraphicalEditPart) iter.next();
    modelToEditPart.put(child.getModel(), child);
}
// Remove child edit parts corresponding to removed model objects
IFavoriteItem[] removed = event.getItemsRemoved();
for (int i = 0; i < removed.length; i++) {

    // Remove the favorites item child if it exists
    IFavoriteItem item = removed[i];
    EditPart child = modelToEditPart.get(item);
    if (child == null)
        continue;
    removeChild(child);

    // Remove the associated resource child if it is not referenced
    Object res = item.getAdapter(IResource.class);
    child = modelToEditPart.get(res);
    if (child == null)
        continue;
    GraphicalEditPart rep = (GraphicalEditPart) child;
    if (rep.getTargetConnections().size() == 0)
        removeChild(child);
}

// Add child edit parts for new model objects
IFavoriteItem[] added = event.getItemsAdded();
for (int i = 0; i < added.length; i++) {

    // Add a favorites item child if necessary
    IFavoriteItem item = added[i];
    EditPart child = modelToEditPart.get(item);
    if (child != null)
        continue;
    child = createChild(item);
    setRandomChildLocation(child);
    addChild(child, getChildren().size());

    // Add a resource child if necessary
    Object res = item.getAdapter(IResource.class);
    child = modelToEditPart.get(res);
    if (child != null)
        continue;
    child = createChild(res);
    setRandomChildLocation(child);
    addChild(child, getChildren().size());
}
}

private void setRandomChildLocation(EditPart child) {
    IFigure childFigure = ((GraphicalEditPart) child).getFigure();
    Random random = new Random();
    int x = random.nextInt() % 150 + 150;

```

```

int y = random.nextInt() % 150 + 150;
childFigure.setLocation(new Point(x, y));
Rectangle constraint = childFigure.getBounds();
LayoutManager layout = getFigure().getLayoutManager();
layout.setConstraint(childFigure, constraint);
}

```

20.6.9 删除模型对象

当收藏夹项通过非GEF Favorites视图从模型删除时，我们的GEF编辑器接收该模型更改事件，并相应地调整图案。但我们想要一种方法，用户可以使用该方法从Favorites GEF Editor自身中触发该删除。该过程的第一部分，关联Edit > Delete命令，已经在20.6.6节中完成了。现在我们必须添加合适的策略和命令以执行实际操作。我们从实现BasicFavoriteItemEditPart中的下列方法开始。

```

protected void createEditPolicies() {
    installEditPolicy(EditPolicy.COMPONENT_ROLE,
        new FavoriteItemComponentEditPolicy());
}

```

然后，实现一个新的编辑策略以接收请求并返回命令。用户可以选择要删除的多个项，而我们的编辑策略必须立即返回一个用于删除和恢复多个项的命令。在该示例中，我们创建一个删除和恢复单个项的命令，然后使用CompoundCommand将这些命令分组。

```

public class FavoriteItemComponentEditPolicy extends ComponentEditPolicy
{
    protected Command createDeleteCommand(GroupRequest request) {
        FavoritesManager manager = ((FavoritesManagerEditPart)
            getHost().getParent()).getFavoritesManager();
        CompoundCommand delete = new CompoundCommand();
        for (Iterator<?> iterator = request.getEditParts().iterator();
            iterator.hasNext();) {
            Object item = ((EditPart) iterator.next()).getModel();
            if (!(item instanceof IFavoriteItem))
                continue;
            delete.add(new FavoriteItemDeleteCommand(manager,
                ((IFavoriteItem) item)));
        }
        return delete;
    }
}

```

最后，删除并恢复单个项的命令。

```

public class FavoriteItemDeleteCommand extends Command
{
    private final FavoritesManager manager;
    private final Object object;

    public FavoriteItemDeleteCommand(
        FavoritesManager manager, IFavoriteItem item) {
        this.manager = manager;
        this.object = item.getAdapter(Object.class);
        setLabel("Delete " + item.getName());
    }
    public void execute() {
        redo();
    }
}

```



```

public void redo() {
    manager.removeFavorites(new Object[] {object});
}

public void undo() {
    manager.addFavorites(new Object[] {object});
}
}

```

请注意上面的命令仅处理模型对象。FavoritesManagerEditPart接收模型更改事件并恰当地更新编辑部分和图案。

20.7 选项板

编辑器中的图案是可移动的、可调整大小的和可删除的，但我们还不能在我们的编辑器中创建收藏夹项。对于图案创建，我们必须设置选项板 (palette) (图20-5) 并添加一个创建工具。当用户在选项板中点击该创建工具，然后点击GEF画布，GEF系统将分发一个创建请求至我们的布局策略 (参见20.6.5节中的FavoritesLayoutEditPolicy)。

20.7.1 创建GEF选项板

修改FavoritesGEFEditor以调用一个创建该选项板的新类。

```

protected PaletteRoot getPaletteRoot() {
    return FavoritesEditorPaletteFactory.createPalette();
}

```

新的FavoritesEditorPaletteFactory创建一个包含一个选择工具、一个选取框和一个收藏夹资源创建工具的选项板 (图20-5)。

```

public class FavoritesEditorPaletteFactory
{
    public static PaletteRoot createPalette() {
        PaletteRoot palette = new PaletteRoot();
        palette.add(createToolsGroup(palette));
        palette.add(createFavoritesDrawer());
        return palette;
    }

    private static PaletteContainer createToolsGroup(PaletteRoot palette)
    {
        PaletteGroup toolGroup = new PaletteGroup("Tools");

        ToolEntry tool = new PanningSelectionToolEntry();
        toolGroup.add(tool);
        palette.setDefaultEntry(tool);

        toolGroup.add(new MarqueeToolEntry());

        return toolGroup;
    }

    private static PaletteContainer createFavoritesDrawer() {
        PaletteDrawer componentsDrawer =
            new PaletteDrawer("Favorites Parts");

        FavoritesCreationFactory factory =
            new FavoritesCreationFactory(FavoriteResource.class);
    }
}

```



```

        CombinedTemplateCreationEntry entry =
            new CombinedTemplateCreationEntry("Favorite Resource",
                "Creates a favorite resource",
                FavoriteResource.class, factory, null,
                null);
        componentsDrawer.add(entry);
        return componentsDrawer;
    }
}

```

这引用了一个包含当用户选择工具然后点击画布时将创建的对象类型的创建工厂。

```

class FavoritesCreationFactory
    implements CreationFactory
{
    private final Class<?> clazz;

    public FavoritesCreationFactory(Class<?> clazz) {
        this.clazz = clazz;
    }

    public Object getNewObject() {
        return clazz;
    }

    public Object getObjectType() {
        return clazz;
    }
}

```

20.7.2 CreateCommand

上面的代码初始化选项板并生成创建请求至编辑策略。现在我们实现FavoritesLayoutEditPolicy中的getCreateCommand方法以创建命令并执行操作。

```

protected Command getCreateCommand(CreateRequest request) {
    if (request.getNewObject() == FavoriteResource.class) {
        FavoritesManager manager = (FavoritesManager) getHost().getModel();
        return new FavoriteItemCreateCommand(manager);
    }
    return UnexecutableCommand.INSTANCE;
}

```

一般地，新的模型对象将由创建过程的getNewObject()方法实例化，并作为创建请求的一部分传递。在这里，我们必须提示用户选择一个收藏夹项，以便我们可以添加该收藏夹至模型，所以我们忽略由创建请求传入的对象。

```

public class FavoriteItemCreateCommand extends Command
{
    private final FavoritesManager manager;
    private Object object = null;

    public FavoriteItemCreateCommand(FavoritesManager favoritesManager) {
        this.manager = favoritesManager;
    }

    public void execute() {
        Shell shell = Display.getCurrent().getActiveShell();
    }
}

```

```

IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
ResourceSelectionDialog dialog =
    new ResourceSelectionDialog(shell, root,
        "Select a resource to add as a favorite item:");
if (dialog.open() == Window.CANCEL)
    return;

Object[] result = dialog.getResult();
if (result.length == 0)
    return;
if (!(result[0] instanceof IFile))
    return;

Object object = result[0];
redo();
}

public void redo() {
    manager.addFavorites(new Object[] { object });
}

public boolean canUndo() {
    return object != null;
}

public void undo() {
    manager.removeFavorites(new Object[] { object });
}
}

```

现在,当用户在选项板中选择Favorite Resource项然后在画布中点击时,执行的命令提示用户在工作区中选择资源(图20-7)。在选择了一个资源后,一个新的收藏夹项添加至模型。FavoritesManagerEditPart接收模型更改事件并正确地更新编辑器。

20.8 总结

一些用户任务可以通过图形用户界面改进,而另一些没有它是不可能的。GEF框架向开发者提供必需的基础结构以添加功能强大的和可视化的、令人感兴趣的内容至程序,以提高用户的效率。

参考文献

本书资源 (2.9节)。

GEF (<http://www.eclipse.org/gef/>)

GEF Articles (<http://www.eclipse.org/gef/reference/articles.html>)

Create an Eclipse-based application using the Graphical Editing Framework(<http://www.ibm.com/developerworks/opensource/library/os-gef/>)

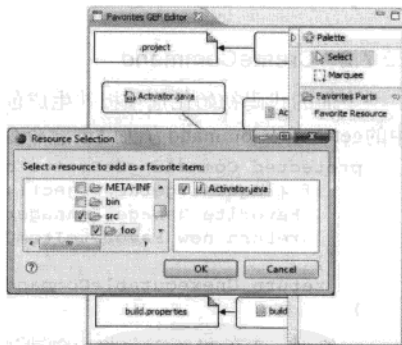


图20-7 收藏夹资源选择对话框

第21章 高级话题

当编写任何书籍时，书末多半存在一些不适合放入任何已有类别的内容。这些内容还不够大或不够多以独立成为一章。本书也不例外。

这一章包含多种需要讨论的主题，而这些主题不适合于放置于本书的其他任何部分。它们包括：

- 高级搜索——引用项目
- 访问内部代码
- 适配器
- 打开浏览器或创建E-mail
- 扩展点中指定的类型
- 修改Eclipse以查找部分标识符
- 标签修饰器
- 后台任务——jobs API
- 插件ClassLoaders
- 早期启动
- 富客户端平台

21.1 高级搜索——引用项目

Eclipse提供了一个优秀的Java搜索功能以用于定位源（参见1.6.2节），而所有搜索的范围被限制为工作区中已载入的项目。如果一个Java类位于插件的JAR文件中，并且该插件不是位于开发项目的Java构建路径，那么该类将不会找到。当创建Eclipse插件时，在搜索范围中包含所有Eclipse插件是有益的，即使是那些没有被任何开发项目所引用的。

为了说明这一点，Eclipse使用Plug-ins视图降低了添加插件至搜索范围的难度（参见1.6.2节）。通过使用它，在开发环境中安装的但没有被工作区中的任意项目引用的插件，可以包含于搜索范围。但是，如果你想要搜索没有在开发环境中安装的插件，如插件的外部库或来源于另一个版本的Eclipse，该怎么办？

一种方法是从Eclipse CVS服务器载入插件项目（参见21.6.1节）。不幸的是，这将占据大量内存，并将你的工作区塞满附加项目。

另一种方法是为每个Eclipse插件创建一个二进制项目。为了创建一个或多个二进制项目，使用Show View对话框（参见2.5节）打开PDE > Plug-ins视图，选择将被作为项目导入的插件，然后右键点击它，并选择Import > As Binary Project。虽然二进制项目比源代码项目占据较少的内存，但它也将你的工作区塞满数百个附加项目。

这里的方法（对于同时搜索和支持多个版本的Eclipse是有用的）是为每个将被搜索的不同版本的Eclipse创建一个引用项目。该项目不包含它自身的源代码，而是将所有Eclipse插件包含于它的classpath，以使一个搜索可以包含Eclipse的完整源代码。为了在你的搜索中包含或不包含一个特定

版本的Eclipse，简单地打开或关闭对应的引用项目。

为了创建一个引用项目，首先创建一个Java项目（参见1.4.1节），然后添加每个Eclipse插件的JAR文件至该项目的Java构建路径（参见1.4.2节）。添加每个插件可能是一个单调的过程，所以将创建一个自动执行该过程的新建项目向导（参见11.2节以了解关于创建向导的特定信息）。

提示 一个创建引用项目的向导可以作为QualityEclipse Tools插件的一部分获得，可以从www.qualityeclipse.com/tools下载。

21.2 访问内部代码

Eclipse将类分为两类：公共API和“仅用于内部使用”。所有位于某个包中的类，并在它的名字中具有“internal”的，对于插件来说是内部的，不应被插件自身之外的任何代码所引用，并有可能在不同版本的Eclipse间有很大区别。所有其他类被认为是公共API，可以从插件外部调用，并遵循严格的Eclipse准则（更改类和方法签名的时间和方法）。当支持多个版本的Eclipse时，遵守公共API是较容易的。

在开发过程中，你可能需要访问对特定Eclipse插件标记为内部的类和方法。每次当你有这样的要求时，你应做的第一件事是复核没有已有的公共API可以完成同样的任务。如果不存在公共API，那么搜索Eclipse.org档案或相关站点（参见A.2节）以了解一个具有你可以使用的解决办法的类似问题。否则，在Eclipse新闻组发布一个消息以描述你的情形并请求建议（参见下一节）。

如果你没有找到解决办法，那么可以在Eclipse Bugzilla跟踪系统发布一个bug报告或功能请求（参见21.2.2节）。在需要时，你可以创建一个片段（参见21.2.6节）以访问必需的代码，除非有一个公共API可用。

21.2.1 Eclipse新闻组

Eclipse新闻组（www.eclipse.org/newsgroups/）向初学者和专家提供了一个共享知识的途径。你将需要一个用户名和密码，所以如果你没有，那么浏览www.eclipse.org/newsgroups/，然后到Request a Password。你提供越多关于你试图完成的任务和展示你已经尝试过的代码，你将越有可能得到回应和你需要的信息。一个模糊的问题将很有可能被忽略或被返回向你询问更多的信息。

做你自己的工作，并且不要期望回答——这些将回答的高手要基于他们的经验，他们对你问题的兴趣和他们的时间。没有人将通过帮助你得到报酬。Eclipse新闻组对于每个人都是开放的。所以请在可能时向他人提供帮助，从而为社区作出贡献。

21.2.2 Bugzilla——Eclipse bug跟踪系统

一旦你已经复核了不存在公共API并且新闻组中没有人对如何完成你的任务提供建议，那么提交一个bug报告或功能请求至Eclipse Bugzilla跟踪系统：

bugs.eclipse.org/

同样地，你需要一个用户名和密码，所以如果你还没有，打开bugs.eclipse.org/，然后选择Create a Bugzilla account。和新闻组一样，你提供越多关于你尝试完成的任务和内容并展示你已经尝试过的工作的代码，Eclipse小组才更可能在以后版本的Eclipse中提供你想要的公共API。为了进一步增加你成功的几率，请确认包含关于你认为应如何修改Eclipse以满足你的需要的细节。或进一步地，你自己做出并与测试用例一起提交更改（参见21.6.1节），以使Eclipse开发小组可以简单地安

装你的更改，测试它们，并继续剩下的工作。你的修改可能包括修改已有Eclipse代码，添加新代码，甚至添加一个新的扩展点（参见17.2节）。

请为你想要修补的bug投票，以使Eclipse小组可以更好地了解哪些更改是重要的，而哪些不是。如同新闻组一样，先完成你自己的工作，并且不要期望你需要的任何内容。Eclipse团队试图满足广泛人群的需要。这是很忙的。

21.2.3 用于访问内部代码的选项

提交请求至Eclipse开发团队将帮助将来版本的Eclipse；但是，要支持当前的和主要的版本该怎么做？有几种技术用于访问内部代码，包括：

- 在某个方法可以公开访问时，直接调用它。
- 在同一个包中创建一个功能类。
- 创建代码至你自己的插件。
- 子类化。
- 使用片段（参见21.2.6节）。

注意 如果你引用内部代码，通过片段直接或间接访问。那么你必须负责在内部代码更改或消失时更改你的代码。

21.2.4 Eclipse的不同之处

Eclipse在插件交互上引入了比一般Java程序更多的限制。每个插件具有它自己的ClassLoader，向通过插件清单指定的插件中的代码限制它的系统可见性（参见2.3.1节）。这意味着即使某个插件中的class A在位于与一个所需要的插件中的class B同样名称的包中，class A将不可以访问class B中的protected和默认方法。Eclipse Java开发环境将如同可以访问这些方法那样对代码进行正确编译，但当代码在Eclipse框架中执行时，插件ClassLoader将限制访问，并抛出一个IllegalAccessException。

这种情形还在库没有由它的插件清单导出时（参见3.3.2节）出现，即使类和方法都被标记为public也将出现。由于你不想要修改一个已有的Eclipse插件，你必须更有策略地应对这些限制。

提示 如果某个第三方将引用并基于你的插件的代码进行构建，那么考虑如同3.3.2节中展示的那样导出你插件中的所有类。你的类可能被用于创建不是由你所最初展望的内容，而隐藏类将阻止他人支持不同的Eclipse版本和代码。显然，在所有可能的地方，通过扩展点的使用提供受控的第三方改进（参见17.1节）。

21.2.5 相关插件

Eclipse 3.1引入了一些经过改进的包级别可见性的目录（x-internal和x-friends）以更精确地定义哪些插件可以访问哪些包。当使用清单编辑器的Runtime页导出包（图2-11）时，使用Package Visibility部分以精确地指定哪些插件可以访问选中包。

比如，在2.8.1节中，你可以向测试和GEF插件限制导出包的可见性。这将生成一个与以下内容相似的Export-Package声明。

```
Export-Package:
    com.qualityeclipse.favorites.handlers;
    x-friends:="com.qualityeclipse.favorites.test,
```

```
com.qualityeclipse.favorites.gef",
com.qualityeclipse.favorites.model;
x-friends:="com.qualityeclipse.favorites.test,
com.qualityeclipse.favorites.gef",
com.qualityeclipse.favorites.views;
x-friends:="com.qualityeclipse.favorites.test,
com.qualityeclipse.favorites.gef"
```

通过这样的方式，其他插件可以保证以一种受控的方式访问内部包。

21.2.6 使用片段

当直接引用代码和复制代码至你自己的插件都无法工作时，你可以尝试使用片段。片段是在一个类插件（plug-in-like）的结构中定义的大量代码。Eclipse将自动将该结构关联至一个已有插件（参见16.3节）。

当考虑Eclipse系统时，由片段添加的代码将与目标插件中已有的代码完全相同地对待。起初，创建片段以插入不同的国家语言支持（National Language Support, NLS）代码至一个基于目标读者的插件中，但你可以利用该机制解决你自己的问题。使用该技术，你不可以覆盖插件中已有的类，但你可以插入新的功能类。这些新的功能类可以用于访问由于具有默认或protected可见性而之前无法访问的方法。

21.3 适配器

Eclipse提供了适配器框架以用于转换一种类型的对象为另一种类型的对应对象。它允许新类型的对象系统地转换为已有类型的Eclipse已知的对象。

当用户选择视图或编辑器中的元素时，其他视图可以请求来源于那些实现了org.eclipse.core.runtime.IAdaptable接口的选中对象的适配对象。这表示Favorites视图选中项可以转换为由已有Eclipse视图请求的对应内容，而不修改任何代码的资源 and Java元素（参见7.4节）。

21.3.1 IAdaptable

对于位于适配器框架的对象而言，它们必须首先实现IAdaptable接口，如同IFavoriteItem那样（参见7.4.2节）。IAdaptable接口包含以下这一个方法以转换一种类型的对象至另一种类型：

- getAdapter(Class)——返回一个作为给定类的实例的对象，并与该对象相关联。如果无法提供这样的对象，则返回null。

IAdaptable接口的实现者尝试提供指定类型的对象。如果它们自己无法转换，那么它们将调用适配器管理器以查看是否存在一个工厂以将它们转换为指定类型。

```
private IResource resource;

public Object getAdapter(Class adapter) {
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager()
        .getAdapter(this, adapter);
}
```

21.3.2 使用适配器

需要转换对象的代码传递想要的类型（如IResource.class）至getAdapter()方法，并且要不获得

一个对应于源对象的IResource的实例，要不获得null以表示这样的转换是不可行的。

```
if (!(object instanceof IAdaptable)) {
    return;
}

MyInterface myObject
    = ((IAdaptable) object).getAdapter(MyInterface.class);

if (myObject == null) {
    return;
}
... do stuff with myObject ...
```

21.3.3 适配器工厂

实现IAdaptable接口允许新类型的对象（如Favorites项）转换为已有类型，如IResource，但如何将已有类型转换为新类型？为了完成该任务，实现org.eclipse.core.runtime.IAdapterFactory接口以将已有类型转换为新类型。

比如，在Favorites产品中，你不可以修改IResource的实现者，但可以实现一个适配器工厂以将IResource转换IFavoriteItem。getAdapterList()方法返回一个数组表示该工厂可以转换的类型，而getAdapter()方法执行转换。在这里，工厂可以转换IResource和IJavaElement对象为IFavoriteItem对象，所以getAdapterList()方法返回一个包含IFavoriteItem.class的数组。

```
package com.qualityeclipse.favorites.model;

import org.eclipse.core.runtime.*;

public class FavoriteAdapterFactory
    implements IAdapterFactory
{
    private static Class<?>[] SUPPORTED_TYPES =
        new Class[] { IFavoriteItem.class };
    public Class<?>[] getAdapterList() {
        return SUPPORTED_TYPES;
    }

    public Object getAdapter(Object object, Class key) {
        if (IFavoriteItem.class.equals(key)) {
            FavoritesManager mgr = FavoritesManager.getManager();
            IFavoriteItem item = mgr.existingFavoriteFor(object);
            if (item == null)
                item = mgr.newFavoriteFor(object);
            return item;
        }
        return null;
    }
}
```

适配器工厂必须在使用之前，在适配器管理器中注册。一般地，插件在启动时在适配器管理器中注册适配器，当插件关闭时，注销它们。比如，在Favorites产品中，添加以下字段至Favorites Activator类：

```
private FavoriteAdapterFactory favoriteAdapterFactory;
```

被添加至FavoritesActivator的start()方法的以下代码注册了该适配器。FavoriteAdapterFactory转换IResource和IJavaElement对象为IFavoriteItem对象，所以你可以在IResource.class中注册适配器为一个参数，然后在IJavaElement.class注册以表示适配器工厂可以从这些类型转换为其他类型。

```
favoriteAdapterFactory = new FavoriteAdapterFactory();
IAdapterManager mgr = Platform.getAdapterManager();
mgr.registerAdapters(favoriteAdapterFactory, IResource.class);
mgr.registerAdapters(favoriteAdapterFactory, IJavaElement.class);
```

此外，必须修改FavoritesActivator的stop()方法以注销该适配器：

```
Platform.getAdapterManager().unregisterAdapters(
    favoriteAdapterFactory);
favoriteAdapterFactory = null;
```

适配器工厂的介绍允许Favorites产品和任意基于它的插件中的代码被更松地耦合至FavoritesManager。比如，不直接调用FavoritesManager类，FavoritesView.pageSelectionChanged()方法（参见7.4.3节）可以使用可适配接口。

```
protected void pageSelectionChanged(
    IWorkbenchPart part, ISelection selection)
{
    if (part == this)
        return;
    if (!(selection instanceof IStructuredSelection))
        return;
    IStructuredSelection sel = (IStructuredSelection) selection;

    List<IFavoriteItem> items = new ArrayList<IFavoriteItem>();
    Iterator<?> iter = sel.iterator();
    while (iter.hasNext()) {
        Object object = iter.next();
        if (!(object instanceof IAdaptable))
            continue;
        IFavoriteItem item = (IFavoriteItem)
            ((IAdaptable) object).getAdapter(IFavoriteItem.class);
        if (item == null)
            continue;
        items.add(item);
    }

    if (items.size() > 0)
        viewer.setSelection(new StructuredSelection(items), true);
}
```

使用适配器工厂具有比直接引用FavoritesManager稍大的系统开销。当你为自己的产品考虑该因素时，你将需要决定更松的耦合的优点是否比额外的复杂性和这种方法稍慢的执行时间显得更重要。

21.3.4 IWorkbenchAdapter

Eclipse使用IWorkbenchAdapter接口显示信息。许多Eclipse视图，如Navigator视图，创建了一个WorkbenchLabelProvider的实例。该标签提供者使用IAdaptable接口将未知的对象转换为IWorkbenchAdapter的实例，然后向该被转换的对象查询可显示的信息，如文本和图像。

对于你的将要在Eclipse视图中显示的对象，比如Navigator，实现IAdaptable接口并返回一个实现IWorkbenchAdapter或扩展WorkbenchAdapter抽象基类的对象。

21.4 打开浏览器或创建E-mail

在你的产品中，你可能需要向用户提供一个简便的方法以访问你的网站或便捷地向你的公司发送一个E-mail。我们首先创建一个按钮。点击该按钮将使用浏览器打开你产品的网页或使用用户的默认E-mail客户端软件创建一个E-mail。简单的方法是使用org.eclipse.swt.program.Program类中的launch()方法（参见15.4.2节），但不幸的是，该方法只能在Windows下起作用。首先，Eclipse提供IWorkbenchBrowserSupport接口以打开一个内部浏览器和一个外部浏览器。然后，创建一个操作以启动默认E-mail客户端软件。

21.4.1 IWorkbenchBrowserSupport

为了使用FavoritesView中的IWorkbenchBrowserSupport，修改OpenWebPageHandler处理器（参见15.4.2节）以创建URL，然后在Eclipse工作台窗口中打开一个浏览器以显示QualityEclipse网站。为了启动一个外部浏览器而不是一个内部浏览器，更改createBrowser()调用的第一个参数为IWorkbenchBrowserSupport.AS_EXTERNAL。

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    // Show a page in an internal web browser
    IWorkbenchWindow window =
        HandlerUtil.getActiveWorkbenchWindow(event);
    IWorkbenchBrowserSupport browserSupport =
        window.getWorkbench().getBrowserSupport();
    URL webUrl;
    try {
        webUrl = new URL("http://www.qualityeclipse.com");
    }
    catch (MalformedURLException e) {
        FavoritesLog.logError(e);
        return null;
    }
    IWebBrowser browser;
    try {
        browser = browserSupport.createBrowser(
            IWorkbenchBrowserSupport.AS_EDITOR, null,
            "Quality Eclipse", "The Quality Eclipse website");
        browser.openURL(webUrl);
    }
    catch (PartInitException e) {
        FavoritesLog.logError(e);
        return null;
    }
    return null;
}
```

如果你在创建RCP程序并且不是基于Eclipse工作台，那么你将需要从org.eclipse.ui.browser插件获取一些功能，并基于在SWT和JFace插件中提供的基本浏览器支持创建。如果你沿着这条路继续往下走，请务必重新查看org.eclipse.ui.browser插件中的DefaultBrowserSupport、WebBrowserEditor和BrowserViewer以及org.eclipse.swt插件中的Browser。

21.4.2 LaunchURL

org.eclipse.help.ui.browser.LaunchURL类提供了另一种打开浏览器的机制。作为org.eclipse.

help.ui插件一部分的该操作代表，可以用于添加工作台菜单（参见6.6.3节）。该工作台菜单将使用浏览器打开一个预定义的网页（通过阅读源代码，该操作似乎具有跨平台支持，但我们仅在Windows上尝试过）。比如，在Favorites产品中，你可以通过在插件清单的“Favorites ActionSet”中添加以下声明向最高级Favorites菜单中添加新操作。

```
<action
    id="com.qualityeclipse.favorites.browseWeb"
    menubarPath="com.qualityeclipse.favorites.workbenchMenu/content"
    label="Browse QualityEclipse"
    icon="icons/web.gif"
    style="push"
    tooltip="Use the LaunchURL class to open a browser"
    class="org.eclipse.help.ui.browser.LaunchURL"
    url="http://www.qualityeclipse.com"/>
```

上面声明中的url属性指定由LaunchURL操作代表显示的网页。不幸的是，插件清单编辑器不支持url属性，所以你必须切换至plugin.xml页以手动编写该属性。此外，似乎也没有与该操作代表等效的一个命令。

21.4.3 OpenEmailAction

org.eclipse.swt.program.Program类中的launch()方法用于在Windows中打开默认E-mail客户端，但在Linux中不能实现同样的效果。你所需要的是一个基于当前平台而分别不同地打开E-mail客户端的单独命令。首先，创建一个新的具有用于标准E-mail元素的字段和设置函数的OpenEmailHandler类。

```
public class OpenEmailHandler extends AbstractHandler
{
    private String recipient;
    private String subject;
    private String body;

    public OpenEmailHandler() {
        setRecipient("info@qualityeclipse.com");
        setSubject("Question");
        setBody("My question is ...\nSecond line\nThird line.");
    }

    public void setRecipient(String recipient) {
        this.recipient = recipient;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public void setBody(String body) {
        this.body = body;
    }
}
```

然后，添加一个确定平台特定模板的execute()方法，填充指定E-mail元素，然后启动E-mail客户端。你可以在以后改进该方法以检查额外的Linux E-mail客户端。

```
public void execute(ExecutionEvent event) throws ExecutionException
{
```

```
String template;
if (SWT.getPlatform().equals("win32")) {
    template = "mailto:${recipient}" +
        "?Subject=${subject}&Body=${body}";
}
else {
    // Put code here to test for various Linux email clients
    template = "netscape mailto:${recipient}" +
        "?Subject=${subject}&Body=${body}";
}

String mailSpec = buildMailSpec(template);

if (mailSpec.startsWith("mailto:")) {
    Program.launch(mailSpec);
}
else {
    try {
        Runtime.getRuntime().exec(mailSpec);
    }
    catch (IOException e) {
        FavoritesLog.logError(
            "Failed to open mail client: " + mailSpec,
            e);
    }
}
return null;
}
```

上面的run()方法调用buildMailSpec()方法以基于提供的平台特定的模板生成一个E-mail说明。它使用它们的不同的值替代模板中的标记,如\${subject}。

```
private String buildMailSpec(String template) {
    StringBuffer buf = new StringBuffer(1000);
    int start = 0;
    while (true) {
        int end = template.indexOf("${", start);
        if (end == -1) {
            buf.append(template.substring(start));
            break;
        }
        buf.append(template.substring(start, end));
        start = template.indexOf("}", end + 2);
        if (start == -1) {
            buf.append(template.substring(end));
            break;
        }
        String key = template.substring(end + 2, start);
        if (key.equalsIgnoreCase("recipient")) {
            buf.append(recipient);
        }
        else if (key.equalsIgnoreCase("subject")) {
            buf.append(subject);
        }
        else if (key.equalsIgnoreCase("body")) {
            appendBody(buf);
        }
    }
}
```



```

        start++;
    }
    return buf.toString();
}

```

buildMailSpec()方法调用appendBody()以在E-mail说明的末尾添加E-mail的内容。回车和换行符被“%0A”替代以在添加E-mail的内容时创建多个行。

```

private void appendBody(StringBuffer buf) {
    if (body == null)
        return;
    int start = 0;
    while (true) {
        int end = body.indexOf('\n', start);
        if (end == -1) {
            buf.append(body.substring(start));
            return;
        }
        if (end > 0 && body.charAt(end - 1) == '\r')
            buf.append(body.substring(start, end - 1));
        else
            buf.append(body.substring(start, end));
        buf.append("%0A");
        start = end + 1;
    }
}

```

现在，你可以添加一个命令（参见6.1节）和菜单添加项以向Favorites视图工具栏添加一个E-mail按钮。

```

<extension point="org.eclipse.ui.commands">
...
<command
    categoryId="com.qualityeclipse.favorites.commands.category"
    defaultHandler=
        "com.qualityeclipse.favorites.handlers.OpenEmailHandler"
    description="Write an email using the default email client."
    id="com.qualityeclipse.favorites.commands.openEmail"
    name="Open Email">
</command>
<menuContribution locationURI="toolbar:
    com.qualityeclipse.favorites.views.FavoritesView?after=additions">
...
<command
    commandId="com.qualityeclipse.favorites.commands.openEmail"
    icon="icons/mail.gif"
    style="push">
</command>

```

这不会发送消息，但将通知E-mail客户端创建具有该特定信息的信息，以使用户可以查看并发送它。上面的代码创建了一个与以下内容类似的E-mail消息：

```

To: info@qualityeclipse.com
Subject: Question

```

```

My question is ...
Second line
Third line.

```

提示 不是所有的系统或浏览器都支持所有mailto选项。为了了解可以被编码至mailto请求的完整列表, 请搜索“mailto语法”。

21.5 扩展点中指定的类型

所有声明扩展点的插件使用`IConfigurationElement.createExecutable()`方法以实例化由其他插件指定的类型(参见17.3.3节)。比如, 考虑以下声明, `org.eclipse.ui`插件将在必须使用`createExecutable()`方法时实例化`myPackage.MyActionDelegate`类。

```
<extension point="org.eclipse.ui.actionSets">
  <action
    label="Open Favorites View"
    icon="icons/sample.gif"
    tooltip="Open the favorites view"
    menubarPath="myMenu/content"
    toolbarPath="Normal/additions"
    id="myProduct.openFavoritesView">
    class="myPackage.MyActionDelegate"
  </action>
</extension>
```

在上面的声明中, 仅指定了完全合格类名, 但还存在一些隐藏功能将在后面的小节中讨论。

21.5.1 参数化的类型

插件清单中指定的类型被使用它们的默认无参数构造函数实例化, 那么如何对它们进行参数化? 比如, 让我们假设在你的菜单中具有两个十分类似的功能。应如何实现这些功能? 一种方法是使用两个不同的操作代表, 每个功能一个。并且该操作代表具有共同的包含所有相同行为的超类。另一种方法是使用一个操作代表, 但以某种方式不同地初始化每个实例, 以执行一个稍微不同的操作, 但该怎么做? 这是这里将要讨论的第二种方法。

参数化类型(在类型的初始化阶段传递附加信息至该类型)通过实现`org.eclipse.core.runtime.IExecutableExtension`接口完成。如果在插件清单中提供了附加信息, 那么Eclipse传递附加信息至使用`setInitializationData`方法的该类型。信息通过`setInitializationData`方法基于插件清单中它的组织方式以不同格式抵达。

1. 非结构化参数

参数化类型的一种方法是在类型的类名称的末尾放置一个冒号, 然后是一个信息字符串。该字符串是非结构化的, 并具有仅需要的信息。Eclipse分析`class`属性, 使用冒号之前的信息确定将初始化的类, 而冒号之后的信息被作为字符串通过`setInitializationData`方法传递至类型。

比如, 在下面的声明中(图21-1), 操作代表`myPackage.MyActionDelegate`将使用它的无参数构造函数实例化, 然后`setInitializationData`方法将使用字符串“one two three”作为它的第三个参数调用。

2. 结构化参数

第二种更结构化的方法是正规地定义参数。不是在一个字符串中定义所有参数, 而是单独每个参数声明为键/值对。每个键/值对放置于一个`java.util.Hashtable`中。该`java.util.Hashtable`将传递至`setInitializationData`方法。

比如, 在下面的`IExecutableExtension`声明(图21-2)中, 操作代表`myPackage.MyAction`

Delegate将使用它的无参数构造函数实例化，然后将使用一个Hashtable作为第三个参数调用setInitializationData方法。Hashtable将包含键/值对“p1”/“one”、“p2”/“two”和“p3”/“three”。这种方法的其他所有内容都与第一种方法相同。

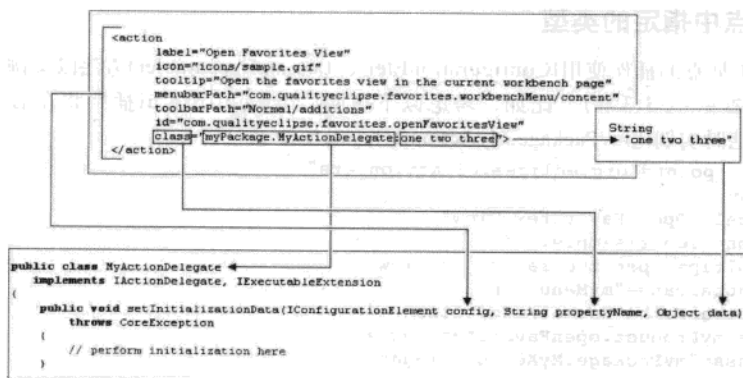


图21-1 具有非结构化参数的IExecutableExtension

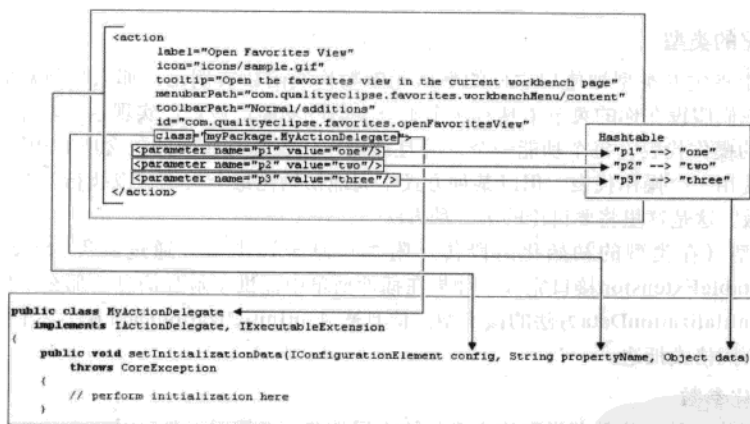


图21-2 具有结构化参数的IExecutableExtension

21.5.2 在不同的插件中引用类

多数情况下，插件清单中引用的类位于该插件中。但是，如果一个插件清单引用了一个包含于JAR文件中或另一个插件中的类该怎么办？默认地，如果指定了完整类名称，那么Eclipse假设位于插件中的类做出声明，所以它不会查找位于不同插件的类。在这种情况下，在类名前加上其他插件的标识符和一个斜杠。

比如，如果plugin.A提供了一个可以被参数化的操作代表类（参见21.5.1节），并且plugin.B提供了一个操作以启动该类的一个参数化的副本，那么plugin.B中的操作代表将与以下内容类似：

```
<action
  id="com.qualityeclipse.favorites.showPartInfo"
  label="Show My View Info"
  menubarPath="myMenu/content"
  class="plugin.A/
    plugin.A.actions.ShowPartInfoActionDelegate">
  <parameter
    name="partClass"
    value="plugin.B.view.myView" />
</action>
```

21.6 修改Eclipse以查找部分标识符

使用插件清单编辑器定义新命令和操作是一个简单的过程。当Eclipse提供了插件探测器（参见2.7.5节），为扩展指定视图和编辑器的上下文菜单查找那些麻烦的标识符（参见6.8节和6.9节）将变得容易许多。简单地悬停于视图或编辑器并打开插件探测器以获取所需的标识符。

在该示例中，让我们假设我们需要使用我们自己的功能通过查询活动的工作台部分以获取该信息。如果它是编辑器或视图，那么转存关于该部分的信息（如上下文菜单标识符）至终端。不幸的是，用于获取这种信息的API不存在（参见21.2节），所以，在创建操作代表之前，你需要修改底层的Eclipse系统以提供合适的访问方法。

21.6.1 修改Eclipse基础

为了修改Eclipse基础，你首先需要从Eclipse库中导出合适的项目，以使你可以在以后创建和提交CVS补丁。提交CVS补丁表示这些更改是如何被反馈至Eclipse提交者，并表达希望能被纳入开发流程的愿望。使用Plug-ins视图（图2-25），你可以通过右键点击并选择Import As > Source Project添加Eclipse插件至工作区。不幸的是，一旦导入，你就不可以创建一个包含你的修改的CVS补丁以提交至Eclipse。作为替代，通过打开CVS Repositories视图（参见1.8.1节）以连接至Eclipse.org开发库，并选择New > Repository Location。在Add CVS Repository对话框中，输入以下值。

- Host——“dev.eclipse.org”
- Repository Path——“/cvsroot/eclipse”
- User——“anonymous”
- Password——留为空白
- Connection Type——“pserver”

当连接上以后，扩展HEAD树节点，找到org.eclipse.ui.workbench项目，并从工作区中导出它（参见1.8.2节）。当从工作区中导出它之后，在Problems视图可以有一些编译错误。这是因为当前使用的Eclipse与编译该项目使用的Eclipse的版本可能不同。插件项目根据其他插件项目的HEAD版本编译，但由于它无法在你的工作区中找到这些其他的插件项目，它将根据当前Eclipse中的插件进行编译。以下是修复该问题的几种不同的办法。

- 查明该插件项目直接或间接以来的每个Eclipse插件项目。
- 下载并安装（但不要启动）最新的Eclipse集成版本，然后使用Plug-in Development > Target Platform将你当前的Eclipse环境重定向至该集成版本中的插件（参见19.2.12节）。缺点是你的工作区中的所有其他插件也将根据该目标Eclipse进行编译。
- 导出一个插件项目的稍早版本。该版本应根据你的Eclipse中包含的插件进行编译。缺点是如

果你编写的任意代码基于在你导出和HEAD版本之间不同的功能，那么当你提交至Eclipse.org时，它可能无法编译。

- 尽你所能地使用上面这些方法的其中一种，然后等待下一个Eclipse里程碑构建版本的发布（它们一般是十分稳定的，而其他版本一般都不是）。下载、安装并根据新版本编写代码并尽可能快地提交你的更改至Eclipse.org。

当载入org.eclipse.ui.workbench项目并且清理了编译错误时，添加以下方法。

```
org.eclipse.ui.internal.PopupMenuExtender
public String getMenuIds() {
    if (staticActionBuilders == null)
        return Collections.EMPTY_SET;
    return staticActionBuilders.keySet();
}

org.eclipse.ui.internal.PartSite
public String[] getContextMenuIds() {
    if (menuExtenders == null)
        return new String[0];
    ArrayList menuIds = new ArrayList(menuExtenders.size());
    for (Iterator iter = menuExtenders.iterator(); iter.hasNext();) {
        final PopupMenuExtender extender =
            (PopupMenuExtender) iter.next();
        menuIds.addAll(extender.getMenuIds());
    }
    return (String[]) menuIds.toArray(new String[menuIds.size()]);
}
```

21.6.2 创建全局操作

接下来，你将创建一个能够使用刚介绍的API的命令。在你选择的插件项目（比如，Favorites插件项目，但不是org.eclipse.ui.workbench插件项目）中，在插件清单中定义一个新的命令（参见6.1节），将其命名为类似于“Show Part Info”，并关联以下处理器。一定要修改插件的classpath以引用工作区中的org.eclipse.ui.workbench项目而不是org.eclipse.ui.workbench外部插件，并确认org.eclipse.ui.workbench位于插件清单中的所需插件列表中。

```
public class ShowPartInfoHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException
    {
        // Determine the active part.

        IWorkbenchPage activePage =
            PlatformUI
                .getWorkbench()
                .getActiveWorkbenchWindow()
                .getActivePage();

        IWorkbenchPart activePart =
            activePage.getActivePart();

        // Search editor references.
```




```
    IEditorReference[] editorRefs =
        activePage.getEditorReferences();
    for (int i = 0; i < editorRefs.length; i++) {
        IEditorReference eachRef = editorRefs[i];
        if (eachRef.getEditor(false) == activePart) {
            printEditorInfo(
                eachRef,
                (IEditorPart) activePart);
        }
    }
}

// Search view references.
IViewReference[] viewRefs =
    activePage.getViewReferences();

for (int i = 0; i < viewRefs.length; i++) {
    IViewReference eachRef = viewRefs[i];
    if (eachRef.getView(false) == activePart) {
        printViewInfo(eachRef, (IViewPart) activePart);
    }
}
return null;
}

private void printEditorInfo(
    IEditorReference editorRef,
    IEditorPart editor) {
    printPartInfo(editorRef, editor);
}

private void printViewInfo(
    IViewReference viewRef,
    IViewPart view) {
    printPartInfo(viewRef, view);
}

private void printPartInfo(
    IWorkbenchPartReference partRef,
    IWorkbenchPart part) {

    println(partRef.getTitle());
    println(" id = " + partRef.getId());
    IWorkbenchPartSite site = part.getSite();
    if (site instanceof PartSite) {
        String[] menuIds =
            ((PartSite) site).getContextMenuIds();
        if (menuIds != null) {
            for (int i = 0; i < menuIds.length; i++)
                println(" menuId = " + menuIds[i]);
        }
    }
}

public void println(String line) {
    System.out.println(line);
}
}
```



21.6.3 测试新功能

创建一个新的启动配置文件（参见2.6节）并启动一个Runtime Workbench以测试该新功能。一定要修改启动配置文件以使它引用工作区中的org.eclipse.ui.workbench项目而不是org.eclipse.ui.workbench外部插件。当你激活编辑器或视图并选择新命令时，你将看到工作台部分的信息出现于Development Workbench的Console视图。

21.6.4 提交更改至Eclipse

在你已经创建Eclipse的一个有用的添加项并认为即使它没有商业价值但它可能真的能帮助其他开发者，你可以将它发布至一个网站，让其他人选择下载并使用。或者更进一步，你可以将它提交至Eclipse.org以通过Bugzilla包含于Eclipse基础中（参见21.2.2节）。

比如，如果你将要提交关于Eclipse基础的更改（参见21.6.1节），你将遵循以下步骤：

1) 使用浏览器打开Eclipse Bugzilla页（bugs.eclipse.org/bugs），并搜索所有已提交项，看看是否其他人已经具有和你的想法类似的想法，并已经报告了一个bug或功能请求（比如，我们已经提交该代码至Bug # 39782）。

2) 如果在搜索后，你已经确定你的添加项还没有由其他人做出，那么打包你的Eclipse基础的更改为一个CVS补丁。为了创建一个补丁以提交至Eclipse.org，选择包含你的修改的Eclipse项目，右键点击它，然后选择Team > Create Patch...。请注意补丁创建功能仅可以用于从库（如dev.eclipse.org，参见21.6.1节）中导出的项目，而不是从导入的二进制或源代码插件项目中导出的项目。

3) 要么创建一个新的bug报告并添加至你补丁的末尾，要么将你的补丁添加至一个已有的bug报告。一定要说明补丁包含内容和为什么你认为它应包含在Eclipse基础代码中。

21.7 标签修饰符

标签修饰符可视化地表示对象的特定属性。比如，如果一个项目存储于库中，那么在Navigator视图中它的文件夹图标右下角具有一个小圆柱。Navigator的标签提供者（参见7.2.5节）返回一个文件夹图像。该文件夹图像随后由库的标签修饰符使用一个小圆柱进行装饰。最终的组合图像显示于Navigator视图中。标签修饰符不限制于只修饰图像，可以通过添加字符至开始和结尾以修改对象的文本。

org.eclipse.ui.decorators扩展点提供了一种机制以添加新的标签装饰符。标签装饰符出现于General > Appearance > Label Decorations首选页（图21-3）并可以由用户选择是否启用。标签修饰符的行为通过实现ILabelDecorator和可选的IFontDecorator和/或IColorDecorator提供。如果用于装饰对象的信息不能立即可用，比如，装饰的类型取决于一个网络查询，那么请实现IDelayedLabelDecorator。

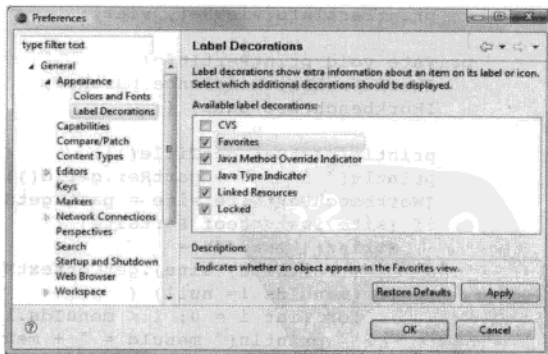


图21-3 标签装饰符首选页

21.7.1 声明标签装饰符

在Favorites产品中，你需要装饰同时出现于Favorites视图和其他视图中的对象。为了完成该任务，创建一个新的具有以下值的org.eclipse.ui.decorators扩展项（参见6.6.1节以了解一个关于创建扩展项的示例）。

- adaptable——“true”

一个表示适配于IResource的类型是否应使用该对象添加项的标记。仅当objectClass适配于IResource时使用该标记。默认值是false。

- class——“com.qualityeclipse.favorites.views.FavoritesLightweightDecorator”

实现org.eclipse.jface.viewers.ILightweightLabelDecorator的类的完全合格名称（参见下一节）。或者，当该装饰符仅具有一个图标而不具有行为时将不特别指明（参见21.7.3节）。

- icon——保持空白

如果装饰符是轻量级的，并且没有指定类，那么这就是覆盖图像以应用的方法（参见下一节）。

- id——“com.qualityeclipse.favorites.favoritesLightweightDecorator”

将用于标识该装饰符的唯一名称。

- label——“Favorites”

将用于General > Appearance > Label Decorations首选项页以代表该装饰符的可转换名称。

- lightweight——“true”

必须为true。重量级的标签装饰符是不建议使用的。

- location——“TOP_LEFT”

应用装饰符图像的位置。默认为BOTTOM_RIGHT。合法的值包括TOP_LEFT、TOP_RIGHT、BOTTOM_LEFT、BOTTOM_RIGHT和UNDERLAY。

- objectClass——“org.eclipse.core.resources.IResource”

该装饰符将应用于的类的完全合格名称。在Eclipse 2.1中由使能嵌套元素所取代，并且不建议使用（参见6.7.2节）。

- state——“true”

一个表示装饰符默认是否是打开的标记。默认值是false。

使用该描述嵌套元素以提供一个关于标签装饰符所做工作的描述：

```
<description>
Indicates whether an object appears in the Favorites view.
</description>
```

如果你想要更明确地指定如何将使用该标签装饰符（参见21.7.3节），那么你可以添加enablement（参见6.7.2节）、and、or和not子元素。比如，使用以下使能表达式替代objectClass属性：

```
<enablement>
  <objectClass
    name="org.eclipse.core.resources.IResource">
  </objectClass>
</enablement>
```

21.7.2 ILightweightLabelDecorator

ILightweightLabelDecorator的实例可以修改对象显示的图像、文本、状态和颜色。当你通过点击class属性值左侧的class标签指定class属性时，创建包含装饰性行为的类。在New Java Class向导

中，选择Generate a new Java class，输入包的名称和类的名称，并点击Finish按钮。

当已经生成了初始类后，确保装饰符实现的是ILightweightLabelDecorator而不是ILabelDecorator。decorate()方法在末尾添加 “[favorite]” 并在所有一个被添加至Favorites视图的资源上覆盖一个绿色的小F。

```
package com.qualityeclipse.favorites.views;
import ...
public class FavoritesLightweightDecorator
    implements ILightweightLabelDecorator, FavoritesManagerListener
{
    private static final String SUFFIX = " [favorite]";
    private static final ImageDescriptor OVERLAY =
        FavoritesActivator.imageDescriptorFromPlugin(
            FavoritesActivator.PLUGIN_ID, "icons/favorites_overlay.gif");
    private final FavoritesManager manager =
        FavoritesManager.getManager();
    public void decorate(Object element, IDecoration decoration) {
        if (manager.existingFavoriteFor(element) != null) {
            decoration.addOverlay(OVERLAY);
            decoration.addSuffix(SUFFIX);
        }
    }
}
```

装饰符还必须在元素的装饰已经更改时通知标签监听器。在这种情况下，无论何时元素已经添加至Favorites视图或从该视图移除，通知监听器相关资源的状态已经更改。这需要注册来源于FavoritesManager的更改事件，然后将这些事件重新广播至所有已注册的ILabelProviderListener实例。

```
private final List<ILabelProviderListener> listenerList =
    new ArrayList<ILabelProviderListener>();
public FavoritesLightweightDecorator() {
    // Make sure that the Favorites are loaded.
    manager.getFavorites();
    manager.addFavoritesManagerListener(this);
}

public void dispose() {
    manager.removeFavoritesManagerListener(this);
}

public void addListener(ILabelProviderListener listener) {
    if (!listenerList.contains(listener))
        listenerList.add(listener);
}
public void removeListener(ILabelProviderListener listener) {
    listenerList.remove(listener);
}

public void favoritesChanged(FavoritesManagerEvent event) {
    Collection<Object> elements = new HashSet<Object>();
    addResourcesTo(event.getItemsAdded(), elements);
    addResourcesTo(event.getItemsRemoved(), elements);

    LabelProviderChangedEvent labelEvent =
        new LabelProviderChangedEvent(this, elements.toArray());
    Iterator<ILabelProviderListener> iter = listenerList.iterator();
```

```

while (iter.hasNext())
    iter.next().labelProviderChanged(labelEvent);
}

private void addResourcesTo(
    IFavoriteItem[] items, Collection<Object> elements)
{
    for (int i = 0; i < items.length; i++) {
        IFavoriteItem item = items[i];
        Object res = item.getAdapter(IResource.class);
        if (res != null)
            elements.add(res);
    }
}

public boolean isLabelProperty(Object element, String property) {
    return false;
}

```

当完成该行为后，所有被添加至Favorites视图的元素将被一个小“F”覆盖，并在视图中具有后缀[favorite]（图21-4）。

21.7.3 装饰性标签装饰符

如果你仅需为了通过在四分之一位置添加一个静态图像来装饰标签，而不需要任意文本修改，那么你可以指定icon属性替代class属性。如果没有指定class属性，Eclipse放置由icon属性指定的图像于由location属性指定的四分之一的的位置。

在这种情况下，没有必要创建一个类以实现ILightweightLabelDecorator，因为Eclipse为你提供这种功能。只读文件装饰符是装饰性标签装饰符的一个示例。

```

<decorator
    lightweight="true"
    location="BOTTOM_LEFT"
    label="Locked"
    icon="icons/locked_overlay.gif"
    state="true"
    id="com.qualityeclipse.favorites.locked">
<description>
    Indicates whether a file is locked
</description>
<enablement>
    <and>
        <objectClass
            name="org.eclipse.core.resources.IResource"/>
        <objectState name="readOnly" value="true"/>
    </and>
    </enablement>
</decorator>

```

在插件清单中使用该声明，一个小锁图标出现于与任意被锁资源相关的图标的左下角（图21-4）。

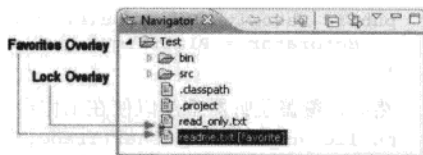


图21-4 具有收藏夹和被锁装饰的导航视图

21.7.4 IDecoratorManager

一旦你已经添加装饰至其他视图, 现在就该装饰你自己的视图了。Eclipse通过IWorkbench中的getDecoratorManager()方法提供了一个DecoratingLabelProvider和一个装饰符管理器。如果视图包含一个简单的列表, 那么你可以和以下内容类似地通过修改FavoritesView createTableViewer()方法使用一个DecoratingLabelProvider封装FavoritesViewLabelProvider:

```
IWorkbench workbench =
    getSite().getWorkbenchWindow().getWorkbench();
viewer.setLabelProvider(new DecoratingLabelProvider(
    new FavoritesViewLabelProvider(),
    workbench.getDecoratorManager()));
```

不幸的是, Favorites视图包含一个表, 所以添加装饰符还需要一些额外工作。我们从添加工作台装饰符至FavoritesViewLabelProvider开始。

```
final IDecoratorManager decorator;
```

```
public FavoritesViewLabelProvider() {
    decorator = PlatformUI.getWorkbench().getDecoratorManager();
}
```

然后, 覆盖监听器方法以使在工作台装饰更改时通知你的视图。

```
public void addListener(ILabelProviderListener listener) {
    decorator.addListener(listener);
    super.addListener(listener);
}

public void removeListener(ILabelProviderListener listener) {
    decorator.removeListener(listener);
    super.removeListener(listener);
}
```

最后修改getColumnText()和getColumnImage()方法在返回请求的文本或图像之前, 分别查询工作台装饰符。

```
public String getColumnText(Object obj, int index) {
    switch (index) {
        case 0: // Type column
            return "";
        case 1: // Name column
            String name;
            if (obj instanceof IFavoriteItem)
                name = ((IFavoriteItem) obj).getName();
            else if (obj != null)
                name = obj.toString();
            else
                name = "";
            String decorated = decorator.decorateText(name, obj);
            if (decorated != null)
                return decorated;
            return name;
        case 2: // Location column
            if (obj instanceof IFavoriteItem)
                return ((IFavoriteItem) obj).getLocation();
            return "";
    }
}
```



```

        default:
            return "";
    }
}
public Image getColumnImage(Object obj, int index) {
    if ((index == 0) && (obj instanceof IFavoriteItem)) {
        Image image = ((IFavoriteItem) obj).getType().getImage();
        Image decorated = decorator.decorateImage(image, obj);
        if (decorated != null)
            return decorated;
        return image;
    }
    return null;
}
}

```

21.8 后台任务——Jobs API

长运行时操作应在后台执行，以使UI可以保持对用户的响应。一种解决办法是分出一个低优先级的线程以执行操作，而不是在UI线程中执行。但是，你如何让用户了解后台操作的进程？Eclipse 提供一个Jobs API以用于创建、关联和显示后台任务。

在Favorites产品中，你需要定期查看是否有新版本可用。为了不打扰用户，你需要让该检查在后台运行，并在操作进行中向用户提供无干扰的进度信息。为了完成该任务，创建NewVersionCheckJob。我们的目标是了解Jobs API，而不是Internet访问，所以NewVersionCheckJob仅模拟一个版本检查。

```

package com.qualityeclipse.favorites.jobs;
import ...
public class NewVersionCheckJob extends Job
{
    private NewVersionCheckJob(String name) {
        super(name);
    }
    protected IStatus run(IProgressMonitor monitor) {
        // Simulate check for new version.
        monitor.beginTask("check for new version", 20);
        for (int i = 20; i > 0; --i) {
            monitor.subTask("seconds left = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Ignored.
            }
            monitor.worked(1);
        }
        monitor.done();
        // Reschedule job to execute in 2 minutes.
        schedule(120000);
        return Status.OK_STATUS;
    }
}

```

用户将通过Favorites首选项页的一个新的单选框控制该操作，所以首先添加一个新的常量至PreferenceConstants（参见12.2.4节）。

```
public static final String
    FAVORITES_NEW_VERSION_CHECK_PREF =
        "favorites.newVersionCheck";
```

然后，通过添加一个新的单选框以在Favorites首选项页显示该新的首选项。这需要一个新字段和createFieldEditors()方法末尾的附加代码（参见12.2.4节）。

```
private BooleanFieldEditor newVersionCheckEditor;

public void createFieldEditors() {
    ... original code here ...
    newVersionCheckEditor = new BooleanFieldEditor(
        PreferenceConstants.FAVORITES_NEW_VERSION_CHECK_PREF,
        "Periodically check for new version"
        + " of Favorites product (simulated)",
        getFieldEditorParent());
    addField(newVersionCheckEditor);
}
```

现在你想要将这个新版本检查任务关联至该首选项，这项任务可以通过添加一个首选项监听器至NewVersionCheckJob实现。首选项监听器根据用户指定的首选项设置计划或取消任务。

```
private static final String JOB_NAME =
    "Favorites check for new version";
private static NewVersionCheckJob job = null;

public boolean shouldSchedule() {
    return equals(job);
}

private static final Preferences preferences =
    FavoritesActivator.getDefault().getPluginPreferences();

private static final Preferences.IPropertyChangeListener
    propertyListener = new Preferences.IPropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent event) {
        update();
    }
};

private static void update() {
    if (preferences.getBoolean(
        PreferenceConstants.FAVORITES_NEW_VERSION_CHECK_PREF))
    {
        if (job == null) {
            job = new NewVersionCheckJob(JOB_NAME);
            // setUser true to show progress dialog
            // or false for system job
            // job.setUser(true);
            job.schedule();
        }
        else {
            if (job != null) {
                job.cancel();
                job = null;
            }
        }
    }
}
```


然后，创建在插件启动和关闭时由FavoritesActivator调用的附加方法。

```
public static void startup() {
    preferences.addPropertyChangeListener(propertyListener);
    update();
}

public static void shutdown() {
    preferences.removePropertyChangeListener(propertyListener);
}
```

当完成这些内容后，在Favorites首选项页选择Periodically check for new version of Favorites product (simulated)单选框将定期执行新版本检查。对用户反馈被作为Jobs API的一部分通过Progress视图（图21-5）自动提供。在Progress视图显示的“% done”基于在beginTask()方法指定的完整任务，而工作单元的数量基于对worked()方法的调用。“seconds left = n”由调用subTask()方法（参见9.4.1节）指定。



图21-5 用于后台操作的进度视图

一般地，任务在后台执行，但IProgressService提供了showInDialog()方法和UIJob类用于在前台执行它们（参见9.4.4节）。此外，如果在任务已经实例化但被计划之前调用setUser(true)，并且如果用户没有选中General > Always run in background首选项，那么它将在前台执行。

```
job = new NewVersionCheckJob(JOB_NAME);
job.setUser(true);
job.schedule();
```

21.9 插件ClassLoader

当你知道你的classpath是正确的（或者在这里是插件清单中的依赖性声明，参见2.3.1节），那么将不被打扰地自动载入类。因此，大部分时间你可以轻松地忽略ClassLoader。但如果你想要载入在编译插件时还不了解的类该怎么办？工作区中由用户开发的代码的信息可以通过JDT接口，比如ICompilationUnit、IType和IMethod访问。然而，它一般不在插件的classpath上，因此不可以执行。一般地（这是个好事情），因为开发的代码可以抛出异常，否则在极少数情况下，它将没有任何提示地使Eclipse崩溃。

Eclipse调试器（参见1.10节）在一个独立的VM中执行用户开发的代码以防止这些问题。但它是重量级的，包括启动一个独立的VM和与它交互以获取结果的内存开销。如果你需要一个快速的方法在和Eclipse同一个VM中执行用户开发的代码，并准备接受这样做的风险，那么你需要编写一个ClassLoader。

为了说明并测试ClassLoader，你首先要声明一个出现于Favorites菜单的新命令（参见6.1节）。

```
<command
    id="com.qualityeclipse.favorites.executeMethod"
    defaultHandler=
        "com.qualityeclipse.favorites.handlers.ExecuteMethodHandler"
    name="Execute Method">
</command>

<menuContribution
    locationURI="menu:org.eclipse.ui.main.menu?after=additions">
```

```

<menu ...>
  ... existing commands ...
  <command
    commandId="com.qualityeclipse.favorites.executeMethod"
    style="push">
  </command>
</menu>
</menuContribution>

```

上面引用的ExecuteMethodHandler（为了了解更多关于处理器的内容，参见6.3节）获取选择的Java方法，载入声明方法的类型，实例化该类型的一个实例，并向Console视图输出执行方法的结果。出于简便性的考虑，选中的方法必须是public且没有参数的。

```

public class ExecuteMethodHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException
    {
        ISelection selection = HandlerUtil.getCurrentSelection(event);
        if (selection instanceof IStructuredSelection)
            System.out.println(
                executeMethod((IStructuredSelection) selection));
        return null;
    }
}

```

execute()方法调用executeMethod()以执行实际操作并返回一个消息，该消息随后被添加至系统控制台。

```

private String executeMethod(IStructuredSelection selection) {
    if (selection == null || selection.isEmpty())
        return "Nothing selected";
    Object element = selection.getFirstElement();
    if (!(element instanceof IMethod))
        return "No Java method selected";

    IMethod method = (IMethod) element;
    try {
        if (!Flags.isPublic(method.getFlags()))
            return "Java method must be public";
    } catch (JavaModelException e) {
        FavoritesLog.logError(e);
        return "Failed to get method modifiers";
    }
    if (method.getParameterTypes().length != 0)
        return "Java method must have zero arguments";

    IType type = method.getDeclaringType();
    String typeName = type.getFullyQualifiedName();
    ClassLoader loader =
        new ProjectClassLoader(type.getJavaProject());
    Class<?> c;
    try {
        c = loader.loadClass(typeName);
    } catch (ClassNotFoundException e) {
        FavoritesLog.logError(e);
        return "Failed to load: " + typeName;
    }
}

```



```
}

Object target;
try {
    target = c.newInstance();
} catch (Exception e) {
    FavoritesLog.logError(e);
    return "Failed to instantiate: " + typeName;
}

Method m;
try {
    m = c.getMethod(method.getElementName(), new Class[] {});
} catch (Exception e) {
    FavoritesLog.logError(e);
    return "Failed to find method: " + method.getElementName();
}

Object result;
try {
    result = m.invoke(target, new Object[] {});
} catch (Exception e) {
    FavoritesLog.logError(e);
    return "Failed to invoke method: " + method.getElementName();
}
return "Return value = " + result;
}
```

ExecuteMethodHandler类使用ProjectClassLoader以载入选中的类至将要执行的Favorites插件。该ClassLoader使用项目的Java构建路径定位类文件，使用标准java.io读取类文件，并使用超类的defineClass()方法在内存中创建类。这还不是完整的。当前它只载入基于源代码的类；从JAR文件或引用项目载入类留给读者作为练习。

```
public class ProjectClassLoader extends ClassLoader {
    private IJavaProject project;
    public ProjectClassLoader(IJavaProject project) {
        if (project == null || !project.exists() || !project.isOpen())
            throw new IllegalArgumentException("Invalid project");
        this.project = project;
    }

    protected Class<?> findClass(String name)
        throws ClassNotFoundException
    {
        byte[] buf = readBytes(name);
        if (buf == null)
            throw new ClassNotFoundException(name);
        return defineClass(name, buf, 0, buf.length);
    }

    private byte[] readBytes(String name) {
        IPath rootLoc = ResourcesPlugin
            .getWorkspace().getRoot().getLocation();
        Path relativePathToClassFile =
            new Path(name.replace('.', '/') + ".class");
        IClasspathEntry[] entries;
```



```

    IPath outputLocation;
    try {
        entries = project.getResolvedClasspath(true);
        outputLocation =
            rootLoc.append(project.getOutputLocation());
    } catch (JavaModelException e) {
        FavoritesLog.logError(e);
        return null;
    }
    for (int i = 0; i < entries.length; i++) {
        IClasspathEntry entry = entries[i];
        switch (entry.getEntryKind()) {
            case IClasspathEntry.CPE_SOURCE :
                IPath path = entry.getOutputLocation();
                if (path != null)
                    path = rootLoc.append(path);
                else
                    path = outputLocation;
                path = path.append(relativePathToClassFile);
                byte[] buf = readBytes(path.toFile());
                if (buf != null)
                    return buf;
                break;
            case IClasspathEntry.CPE_LIBRARY:
            case IClasspathEntry.CPE_PROJECT:
                // Handle other entry types here.
                break;
            default :
                break;
        }
    }
    return null;
}

private static byte[] readBytes(File file) {
    if (file == null || !file.exists())
        return null;
    InputStream stream = null;
    try {
        stream =
            new BufferedInputStream(
                new FileInputStream(file));
        int size = 0;
        byte[] buf = new byte[10];
        while (true) {
            int count =
                stream.read(buf, size, buf.length - size);
            if (count < 0)
                break;
            size += count;
            if (size < buf.length)
                break;
            byte[] newBuf = new byte[size + 10];
            System.arraycopy(buf, 0, newBuf, 0, size);
            buf = newBuf;
        }
        byte[] result = new byte[size];
    }
}

```



```

        System.arraycopy(buf, 0, result, 0, size);
        return result;
    }
    catch (Exception e) {
        FavoritesLog.logError(e);
        return null;
    }
    finally {
        try {
            if (stream != null)
                stream.close();
        }
        catch (IOException e) {
            FavoritesLog.logError(e);
            return null;
        }
    }
}
}
}

```

21.10 早期启动

根据3.4.2节中描述的，使用org.eclipse.ui.startup扩展点以确保你的插件将在Eclipse启动时一起启动。不要悄悄地这样做，因为它违反了Eclipse惰性载入机制，这是由于Eclipse总是载入并执行你的插件，从而消耗了宝贵的内存和启动时间。如果你必须这么做，那么请保持你的早期启动插件是比较小的，以使它在启动时占用较少的内存并能快速执行。

21.10.1 管理早期启动

Eclipse没有提供一种在程序中指定当Eclipse启动时，是否应立即启动某个插件的机制。如果你有一个或多个需要早期启动的插件，那么请考虑创建一个管理早期启动的小插件（图21-6）。比如，如果你有一个大型插件，并且仅在用户选中了一些功能时，它才需要早期启动，那么创建一个小的早期启动插件以确定这些功能是否已经被选中，并且如果已经被选中，那么启动该大型插件。

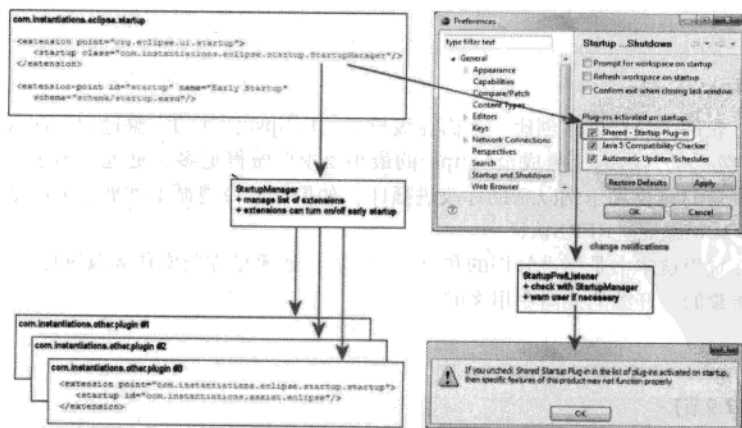


图21-6 用于管理其他插件的早期启动的插件

21.10.2 取消早期启动

用户可以使用首选项General > Startup and Shutdown页取消早期插件启动。如果你已经添加了一个早期启动扩展项,那么你的插件将会出现在该列表中,用户可以取消它的启动。你可以检测到该情况并警示用户你的插件的功能将会打折扣。对于FavoritesActivator,我们添加以下内容。

```
public static boolean isEarlyStartupDisabled() {
    String plugins = PlatformUI.getPreferenceStore().getString(
        /*
         * Copy constant out of internal Eclipse interface
         * IPreferenceConstants.PLUGINS_NOT_ACTIVATED_ON_STARTUP
         * so that we are not accessing internal type.
         */
        "PLUGINS_NOT_ACTIVATED_ON_STARTUP");
    return plugins.indexOf(PLUGIN_ID) != -1;
}
```

21.11 富客户端平台

即使Eclipse是从颂歌“一个可以用于任何用途但没有特别之处的开放、可扩展IDE”(来源于www-128.ibm.com/developerworks/opensource/library/os-plat)中开始它的生命的,但它不仅如此。Eclipse平台的第一个主要重构包括从IDE特定的元素分离通用工具元素,以使它成为一个不仅只适用于Java开发的框架。富客户端平台代表Eclipse平台的第二个主要重构,从通用工具元素分离应用程序基础架构,如视图、编辑器和透视图。在该最近的调整结构之后,Eclipse IDE基于一个通用工具框架,而该框架又基于一个被称为富客户端平台(Rich Client Platform, RCP)的通用应用程序框架。

RCP的核心由一小部分提供基础应用程序框架,包括操作集、透视图、视图和编辑器的插件或包组成。它不包含任何工具或IDE特定的内容,如源代码编辑、重构、编译或构建。任意工具和IDE特定的插件可以作用RCP程序的一部分使用,但不是必需的。到目前为止讨论过的所有内容,从操作和视图到备忘单和jobs是创建RCP程序的一部分,但还有更多内容。《Eclipse Rich Client Platform》(McAffer and Lemieux 2005),是本书的最佳配合读物,提供了RCP程序特定的附加细节。

21.12 总结

纵观本书,我们已经提供了创建一个商业级质量的Eclipse插件的完整过程的深入讨论。对于我们而言,“商业级质量”表示比集成至Eclipse的最小要求要做得更多、更远。出于该目的,我们已经尝试了展示大量的建议和示例以帮助你改进插件。如果你已经遵循本书展示的准则,你还将能较好地提交插件至IBM以通过RFRS认证。

我们希望你觉得这本书是增进知识的和有用的。我们还希望你将它作为改进你创建插件的指南,无论它们是高质量的、开源的或商业用途的。

参考文献

本书资源(2.9节)。

Krish-Sampath, Balaji, “Understanding Decorators in Eclipse,” IBM, January 16, 2003

(www.eclipse.org/articles/Article-Decorators/decorators.html).

Valenta, Michael, "On the Job: The Eclipse Jobs API," IBM, September 20, 2004 (www.eclipse.org/articles/Article-Concurrency/jobs-api.html).

Erickson, Marc, "Working the Eclipse Platform," IBM, November 1, 2001 (www-128.ibm.com/developerworks/opensource/library/os-plat).

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*, Addison-Wesley, Boston, 2005.



附录A Eclipse插件和资源

Eclipse的广泛使用和改进催生了一个新的行业，即为Eclipse添加新的功能。在本书写作时，有超过1000个插件可以用于扩展Eclipse可以想到的所有方面。这些插件从高质量、商业化和开源到相对质量较低的实验版本。

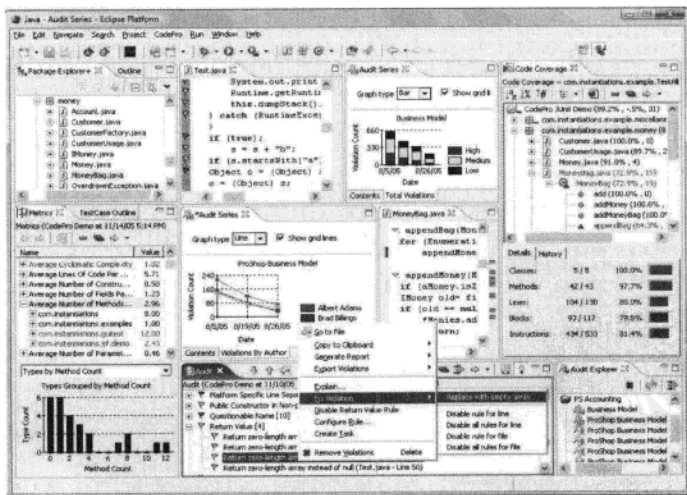
在使用Eclipse的过去几年里，我们已经发现（并且有一些是我们自己创建的）一些十分有用的Eclipse插件。

A.1 插件

插件的下列列表（一些是商业的，一些是开源的，一些是昂贵的，一些是便宜的或免费的）表示你应仔细看一看的插件的简单列表。它们所有都是非常高质量的，并在Eclipse社区获得很高的评价。

A.1.1 CodePro Analytix

CodePro Analytix（1299美元，一个非商业的绑定版本是99美元）添加超过500个对Eclipse的改进，包括最佳做法、分析、测试、可用性和协作。



URL: www.instantiations.com/codepro/analytix/

主要功能包括代码审计、度量（metrics）、Javadoc修复、JUnit测试生成、代码覆盖、重复代码分析、设计模式、依赖性分析器、任务计划和团队合作工具。

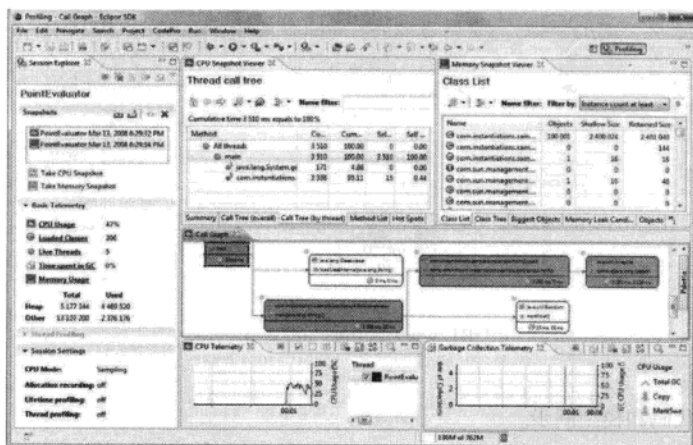
- 代码审计可以捕获超过950个审计冲突，并完整支持Ant脚本和无头操作。动态代码审计模式在错误发生时捕获，而内建的“快速修复”集成将自动修复大部分冲突。可以通过Eclipse扩

展点和与其他开发者交换审计规则集轻松地添加你自己的审计规则。可以生成多种格式的详细审计报告。

- 代码规格具有向下发掘的功能和触发点。
- 重复代码分析器可以查找多个项目间被复制和粘贴的代码。
- 项目/包依赖性分析器通过图表检查循环和闭包。它可以生成详细的依赖性报告和规格。
- Javadoc修复工具。
- 用于注释、标识符、字母、属性和XML文件的拼写检查器。
- 自动JUnit测试用例生成和JUnit测试编辑。
- 代码覆盖分析。
- 强大的Eclipse脚本任务计划器。
- 首选项导入/导出/交换工具。

A.1.2 CodePro Profiler

CodePro Profiler (499美元) 是一个基于Eclipse的软件产品。它可以让Java开发者在开发中轻松地查找程序代码中的性能问题。



URL: www.instantiations.com/codepro/profiler/

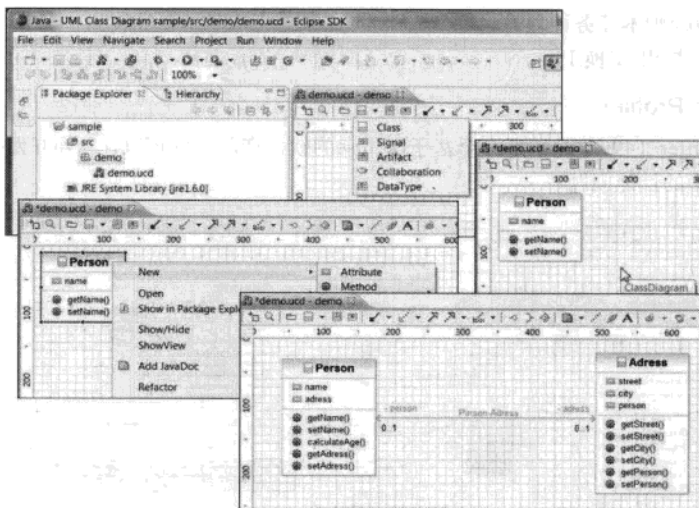
主要功能包括：

- 遥测 (Telemetry) 视图 (CPU、内容、垃圾收集等)
- CPU采样
- CPU字节码仪表
- 内存、线程和监视器分析
- 死锁检测
- 分配跟踪
- 对象生命周期分析
- 内存与CPU快照

- 强大的过滤功能
- 导入/导出快照
- 快照比较
- 灵活的报告创建
- 与Eclipse、Rational和MyEclipse的无缝集成

A.1.3 EclipseUML

EclipseUML是一个可视化建模工具，与Eclipse和CVS能良好集成。它可以管理数百个并发连接，因此适合于大型软件开发项目。



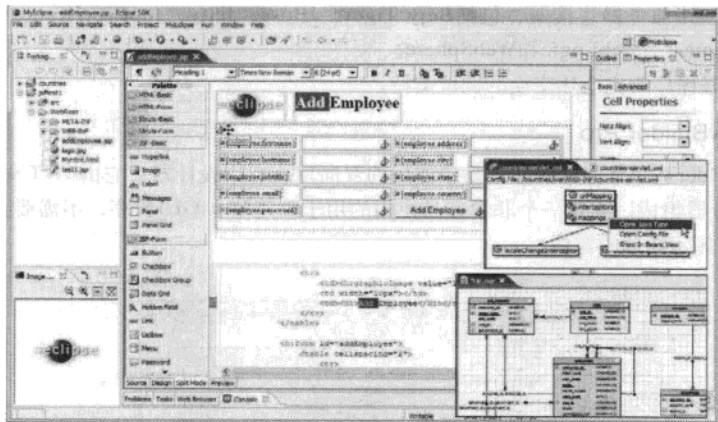
URL: www.eclipseuml.com/

主要功能包括：

- 活动的双向代码和模型同步。
 - 原生的GEF集成。
 - 原生的EMF (Eclipse Modeling Framework) 集成。
- 企业版（1990美元）添加了以下功能：
- 完整Java Version 2 Enterprise Edition (J2EE) 生命周期。
 - 完整数据库建模生命周期。
 - 正添加开放的API和UML分析，并将提供独特的机会以完全自定义程序。

A.1.4 MyEclipse Enterprise Workbench

MyEclipse Enterprise Workbench（标准版29美元/年，专业版49美元/年）是一个Eclipse的完整产品扩展。它提供了一个基于Eclipse平台的全功能的J2EE IDE。MyEclipse支持JSP（JavaServer Pages）、EJB（Enterprise JavaBeans）、XML、JSF、AJAX和Struts完整开发周期（编写代码、部署、测试和调试）。



URL: www.myeclipseide.com/

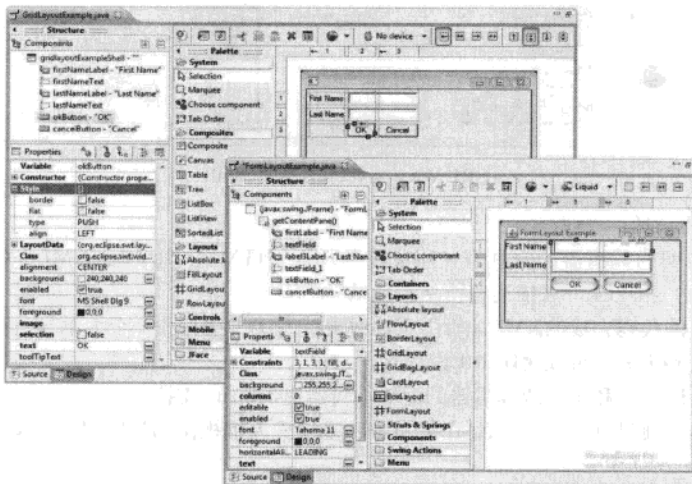
它的主要功能包括:

- 具有一个聪明的编辑器,它具有代码完成和JSP、HTML、Struts、XML、CSS (Cascading Style Sheets) 和J2EE部署描述符语法彩色显示。
- 它具有聪明的代码完成的XML编辑器用于离线支持的DTD缓存和一个大纲导航查看器。
- 具有往返代码生成的可视化HTML设计器。
- 通过具有代码完成和布局流查看器的配置编辑器提供Struts支持。
- 具有图形导航流设计器、高级XML编辑器和多模式大纲视图的JSF (Java Server Faces) 开发软件。
- 具有效率工具、代码生成、DB浏览器连接器集成的Hibernate开发工具。
- 数据库浏览器和SQL编辑器。
- JSP语法验证和本地JSP调试,以及对JSR045的完整支持。
- 对于包含的JSP文件和JSP呈现的步进调试。
- 对于JSP 2.0表达式语言的支持。
- 用于JSP、HTML、XML、Servlet和Applet的可自定义创建模板。
- 实时呈现的集成浏览器。
- Spring IDE集成。
- Tapestry、XDoclet和J2EE 1.4的支持。
- GIF、JPG、BMP、PNG和ICO图像的特别图像预览。
- Web、EAR和EJB项目的创建。
- Java项目至Web项目的转换。
- WAR、JAR、EAR的导入和导出。
- EJB向导。
- 程序的按需同步以及至集成服务器的自动部署。
- 基于Archive的部署 (EAR和WAR)。
- 开始与停止服务器的集成控制。

- 超过20个应用服务器连接器, 包括Bejy Tiger、JBoss、Jetty、Jonas、JRun、Oracle、Orion、Resin、Tomcat、WebLogic和WebSphere。
- 对于已部署程序的完全热交换的调试支持。

A.1.5 WindowBuilder Pro

WindowBuilder是一个强大的并且易于使用的双向Java GUI设计器。它由SWT设计器、Swing设计器和GWT设计器组成。它是一个非常容易使用的用于创建Java GUI程序, 不需要花费大量的时间用于编写代码以显示简单的窗体。



URL: www.windowbuilderpro.com/

免费版的主要功能包括:

- 通过拖放复合组件、布局和控件实现所见即所得 (what you see is what you get, WYSIWYG) GUI编辑原生SWT和Swing控件。
- 当实现为一个双向工具时, WindowBuilder直接生成Java代码。这些Java代码可以在图形编辑器中更改或直接在源代码中更改。所有直接对源代码做出的更改将在图形编辑器中体现。
- 仅适用纯SWT和Swing类, 运行时不产生任何额外开销。不会添加任意特殊库至项目。
- 包含一个方便的属性编辑器以用于简单和直观的属性编辑。所有更改将立即显示于代码编辑器和图像编辑器中。
- 显示一个组件树, 使得在组件间的导航容易很多。
- 包含SWT程序和JFace对话框创建向导。
- 对于所有SWT和Swing控件的完整支持。
- 对于SWT网格、填充和行布局管理器及Swing边界、流、网格、卡片和盒式布局管理器的完整支持。
- 与Eclipse工作台的无缝集成。仅需要将其解压缩并重启Eclipse。
- 一直运行的不需要编译的UI测试, 通过点击一个按钮就可以实现。

主要功能包括：

- 点击并记录——快速并简单地记录GUI测试。如同正常的那样与你的程序交互并让WindowTester记录你的操作并自动生成测试用例。这些生成的测试是纯Java的，并可以使用所有Java语言的功能进行自定义。
- 自动回放——自动使用程序的GUI。
- 连续测试——测试用例执行可以容易地集成一个连续构建系统，以使在每一次构建你的程序时测试它的正确性。
- 代码覆盖——理解程序的哪些部分在执行一个或多个测试时使用。
- 富GUI测试库——隐藏GUI测试执行的复杂性和线程事务并降低测试用例自定义的难度。

A.2 资源

以下是一个Eclipse相关插件、项目和信息的Web链接的简单列表。首先列出的是Eclipse.org网站。它应成为你了解Eclipse相关信息的第一站。随后是一些较大的提供Eclipse相关信息和插件列表的站点。最后是一些包含值得我们关注的不同质量的插件、项目、笔记和信息的站点。

A.2.1 Eclipse.org

www.eclipse.org/——开始了解Eclipse和Eclipse相关技术的地方。可以从这里获取下载、文档、技术文章、邮件列表等。Eclipse.org的主项目被进一步划分为以下子项目：

- The Eclipse Project——主平台和它的工具
- Platform——框架和通用服务
- JDT——Java开发工具
- PDE——插件开发环境
- Equinox——一个OSGi框架
- Eclipse Tools Project——Eclipse的二级工具
- CDT——C/C++开发工具
- GEF——图形用户框架
- COBOL——完整功能的COBOL IDE
- EMF——用于生产模型的Java/XML框架
- VE——Visual editor；用于创建GUI构建器的框架
- UML2——UML 2.0建模工具框架
- The Eclipse Web Tools Platform (WTP)——J2EE Web程序
- WST——Web标准工具
- JST——J2EE标准工具
- JSF——JavaServer Faces工具
- Test & Performance Tools (TPTP)——测试与性能
- TPTP平台
- 监视工具
- 测试工具
- 跟踪与分析工具



- 商业信息与报告工具项目 (The Business Intelligence and Reporting Tools Project, BIRT)
- Eclipse数据工具平台项目
- 设备软件开发平台 (Device Software Development Platform, DSDP)
- The Eclipse Technology Project——基于Eclipse的技术孵化器
- AJDT——AspectJ开发工具项目
- ECESIS——Eclipse社区教育项目
- ECF——Eclipse交流框架
- 生产模型转换器 (Generative Model Transformer) ——用于模型驱动软件开发的工具

Eclipse.org网站还具有一个社区页面 (www.eclipse.org/community/index.html)。它列出了即将出现的事件、课程和包含Eclipse更多信息的相关网站的链接。

A.2.2 Eclipse插件中心

www.eclipseplugincentral.com——一个致力于通过帮助开发者定位、评估和获取插件支持Eclipse社区的不断成长的站点。这些插件用于帮助开发者更快、更好和更经济地分发他们的项目。Eclipse插件中心 (Eclipse Plug-in Central, EPiC) 通过提供关于产品和服务的市场更新、评论、评分、新闻、论坛、社区列表增加了其价值, 并支持Eclipse基金会。

A.2.3 Eclipse wiki wiki

eclipse-wiki.info——一个包含关于从新闻组、邮件列表和类似地方收集的Eclipse信息的wiki wiki站点。该站点的可用性是无法预测的, 但当可用时, 它包含了丰富的信息。

A.2.4 Planet Eclipse

planetecclipse.org/planet——Eclipse新闻和博客聚合器。该站点是与所有Eclipse相关内容保持同步的最佳途径。

A.2.5 EclipseCon

www.eclipsecon.org——关于EclipseCon技术大会的信息的网站。

A.2.6 Eclipse复活节彩蛋

mmoebius.gmxhome.de/eclipse/eastereggs.htm——一个关于Eclipse的链接和资源的列表。还有一个关于Eclipse复活节彩蛋的用于娱乐的页面。

A.2.7 IBM Alphaworks on Eclipse

www.alphaworks.ibm.com/eclipse——一个包含来源于IBM Alphaworks实验室的Eclipse相关技术和文章的站点。

A.2.8 IBM Eclipse research

www.research.ibm.com/eclipse/——一个包含关于基于Eclipse技术的IBM授权和程序的信息的来源。

A.2.9 QNX Momentics

www.qnx.com/products/ps_momentics/——一个基于Eclipse的用于编写关于实时系统的代码的IDE。

附录B Ready for Rational Software

IBM Rational软件开发平台（SDP）是来源于IBM的一个开发的、综合的、多语言的WebSphere开发环境。Rational软件家族的三个主要产品：Web Developer、Application Developer和Software Architect。还有许多工具集为WebSphere和其他IBM中间件的特殊组件提供工具（比如，Voice Toolkit for WebSphere Studio）。用户根据他们程序中使用的WebSphere平台和其他中间件来混合使用核心产品和工具集。产品和工具集被Microsoft Windows和Linux系统所支持。

Rational SDP基于开源的Eclipse工具平台。该平台提供了一个完整可扩展性基础结构。来源于IBM商业伙伴的Eclipse插件工具产品因此可以紧密集成至SDP。并且可以使用商业伙伴工具的独特功能扩展该SDP。

然而，对于要从商业伙伴插件获取价值的用户而言，插件必须安全地安装至Rational软件，并与Eclipse平台和其他插件具有良好的互操作性。插件还必须支持通用的Eclipse/Rational Software UI和通用Eclipse/Rational行为。

RFRS软件验证项目为通过Eclipse平台和组件集成至SDP的插件定义了集成条件。遵循这些条件可以帮助确保你插件的用户满足SDP的安装、互操作性和UI标准。

RFRS软件验证项目是IBM家族的“Ready for”技术验证项目（www.ibm.com/partnerworld/isv/rational/readyfor.html）的一部分。任意作为PWD（Partner World for Developers）成员的IBM商业伙伴可以验证插件产品与RFRS条件的满足度并加入RFRS项目。商业伙伴可以随后在任意已验证的产品中使用“Ready for IBM Rational Software”标记以作为产品满足集成条件的验证的可见认证。标记的使用使得商业伙伴将他们的RFRS-validated产品与他们的竞争产品区分开来，并通过增强的用户信息和降低用户评估时间缩短销售周期。

RFRS也是通向多种不断增加的共同营销资源的途径。这些资源可以帮助项目成员扩展他们的市场范围。这些优点包括在商业展示和IBM路线展示中的RFRS展示，在RFRS插件中心网站（www.ibm.com/developerworks/websphere/downloads/plugin/）中的已验证产品中列出，并在新闻邮件和IBM销售flash中的商业伙伴内容中出现。

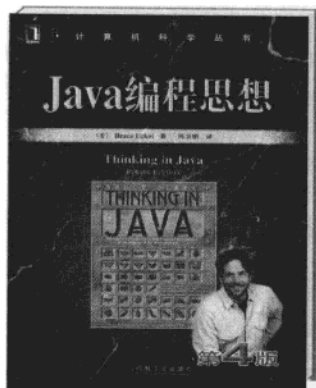
商标

下列内容是IBM公司在美国或其他国家，或在美国和其他国家共同注册的商标：

- IBM
- Rational
- PartnerWorld
- WebSphere

Microsoft和Windows是微软公司在美国或其他国家，或在美国和其他国家共同注册的商标。其他公司、产品和服务名称可能是其他的商标或服务标记。

好书推荐



作者: Bruce Eckel
书号: 978-7-111-21382-6
定价: 108.00元



作者: Joshua Bloch
书号: 978-7-111-25583-3
定价: 52.00元



作者: Gary Cornell
书号: 978-7-111-23950-5
定价: 98.00元



作者: Gary Cornell
书号: 978-7-111-25611-3
定价: 118.00元



专业成就人生
立体服务大众

www.hzbook.com

填写读者调查表 加入华章书友会 获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：

书号：7-111-()

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐ 朋友推荐 ☐ 书店 ☐ 图书目录 ☐ 杂志、报纸、网络等 ☐ 其他

2. 您从哪里购买本书：

☐ 新华书店 ☐ 计算机专业书店 ☐ 网上书店 ☐ 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

☐ 是，我的计划是_____ ☐ 否

7. 您希望获取图书信息的形式：

☐ 邮件 ☐ 信函 ☐ 短信 ☐ 其他_____

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收
邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzjsj@hzbook.com

[G e n e r a l I n f o r m a t i o n]

书名= ECLIPSE 插件开发 原书第 3 版

SS 号= 1 2 5 7 6 3 7 8